MEGA 65

USER'S GUIDE

MUSEUM OF ELECTRONIC GAMES & ART

MEGA65 TEAM

Dr. Paul Gardner-Stephen
(highlander)
Founder
Software and Virtual Hardware Architect
Spokesman and Lead Scientist

Detlef Hastik
(deft)
Co-Founder
General Manager
Marketing & Sales

Martin Streit
(seriously)
Video and Photo Production
Tax and Organization
Social Media

Anton Schneider-Michallek
(adtbm)
Hardware Pool Management
Soft-, Hard- and V-Hardware Testing
Forum Administration

Falk Rehwagen
(bluewaysw)
Jenkins Build Automation
GEOS, Hardware Quality Management

Antti Lukats
(antti-brain)
Host Hardware Design and Production

Dieter Penner
(doubleflash)
Host Hardware Review and Testing
File Hosting

Dr. Edilbert Kirk
(Bit Shifter)
Manual and Tools
ROM Enhancements

Gábor Lénárt
(LGB)
Emulator

Mirko H.
(sy2002)
Additional Hardware and Platforms

Farai Aschwanden
(Tayger)
File Base, Tools
Financial Advisory

Thomas Hertzler
(grumpyninja)
USA Spokesman
Artist Relations

Andrew Owen
(Cheveron)
Keyboard Advisory, Sinclair Support

Daniel England
(Mew Pokémon)
Additional Code and Tools

Roman Standzikowski
(FeralChild)
Open ROMs

Hernán Di Pietro
(indiocolifa)
Additional Emulation

Reporting Errors and Omissions

This book is being continuously refined and improved upon by the MEGA65 community. The version of this edition is:

```
commit 9d491c47b1742ce405c3b15ea8b1a58d0cd0cb6a
date: Thu Nov 4 18:12:21 2021 +0100
```

We want this book to be the best that it possibly can. So if you see any errors, find anything that is missing, or would like more information, please report them using the MEGA65 User's Guide issue tracker:

`https://github.com/mega65/mega65-user-guide/issues`

You can also check there to see if anyone else has reported a similar problem, while you wait for this book to be updated.

Finally, you can always download the latest versions of our suite of books from these locations:

- `https://mega65.org/mega65-book`
- `https://mega65.org/user-guide`
- `https://mega65.org/developer-guide`
- `https://mega65.org/basic65-ref`
- `https://mega65.org/chipset-ref`
- `https://files.mega65.org/manuals-upload`

MEGA65 USER'S GUIDE

Published by
the MEGA Museum of Electronic Games & Art e.V., Germany.

1st Edition

Copyright ©2019 - 2021 by Paul Gardner-Stephen, the MEGA Museum of Electronic Games & Art e.V., and contributors.

This user guide is made available under the GNU Free Documentation License v1.3, or later, if desired. This means that you are free to modify, reproduce and redistribute this user guide, subject to certain conditions. The full text of the GNU Free Documentation License v1.3 can be found at `https://www.gnu.org/licenses/fdl-1.3.en.html`.

Implicit in this copyright license, is the permission to duplicate and/or redistribute this document in whole or in part for use in education environments. We want to support the education of future generations, so if you have any worries or concerns, please contact us.

November 4, 2021

Contents

1 Introduction **ix**

Welcome to the MEGA65! xi

Other Books in this series xii

Come Join Us! xii

2 Setup **1**

Unpacking and Connecting the MEGA65 3

Rear Connections 3

Side Connections 4

MEGA65 Screen and Peripherals 5

Optional Connections 6

Replacing the Real-Time Clock Battery 6

Operation 7

Using the MEGA65 7

The Cursor 8

3 Getting Started **9**

Keyboard 11

Command Keys 11

Function Keys 14

The Screen Editor 15

Editor Functionality 18

4 Configuring your MEGA65 **21**

Important Note 23

Formatting SD cards 23

Installing ROM and Other Support Files 26

ROM File 27

Support Files 28

On-boarding 28

Configuration Utility 31

Input Devices 34

Chipset 35
Video 36
Audio 36
Network 37

5 C64, C65 and MEGA65 Modes 39
Switching Modes from BASIC 41
From MEGA65/C65 to C64-mode 41
From C64 to MEGA65/C65-mode 42
Entering Machine Code Monitor Mode 42
The KEY Register 42
Exposing Extra C65 Registers 43
Disabling the C65/MEGA65 Extra Registers 44
Enabling MEGA65 Extra Registers 44
Traps to Look Out For 45
Accessing Memory from BASIC 65 45
The MAP Instruction 46

6 Cores and Flashing 47
What are Cores, and Why Do They Matter? 49
Bitstream files 49
File types 49
Where to Download 50
Selecting a Core 51
Installing an Upgrade Core for the MEGA65 52
Installing Other Cores 54
Creating Cores for the MEGA65 54
Replacing the Factory Core in Slot 0 55
Understanding The Core Booting Process 55
DIP Switches 57

7 Floppy Disks, the Freezer, and D81 Images **59**
Disk Drives 61
Disk Drive Terminology 61
The Freezer 62
Using D81 images 64
Auto Booting From a Disk 65
Saving D81 file settings 66

APPENDICES **67**

A BASIC 65 Command Reference **67**
Commands, Functions and Operators 69
BASIC Command Reference 75

B Special Keyboard Controls and Sequences **215**
PETSCII Codes and CHR$ 217
Control codes 219
Shifted codes 222
Escape Sequences 222

C Supporters & Donors **227**
Organisations 229
Contributors 230
Supporters 231

INDEX **237**

CHAPTER 1

Introduction

- Welcome to the MEGA65!
- Other Books in this series
- Come Join Us!

WELCOME TO THE MEGA65!

Congratulations on your purchase of one of the most long-awaited computers in the history of computing! The MEGA65 is community designed, and based on the never-released Commodore® 65[1] computer; a computer designed in 1989 and intended for public release in 1990. Decades have passed, and we have endeavoured to invoke memories of an earlier time when computers were simple and friendly. They were not only simple to operate and understand, but friendly and approachable for new users.

These 1980s computers inspired many of their owners to pursue the exciting and rewarding technology careers they have today. Just imagine the exhilaration these early computing pioneers experienced, as they learned they could use their new computer to solve problems, write a letter, prepare taxes, invent new things, discover how the universe works, and perhaps even play an exciting game or two! We want to re-awaken that same level of excitement (which alas, is no longer found in modern computing), so we have created the **MEGA65**.

The MEGA65 team believes that owning a computer is like owning a home. You don't just use a home; you change things, big and small, to make it your own custom living space. After a while, when you settle in, you may decide to renovate or expand your home to make it more comfortable, or provide more utility. Think of the MEGA65 as your very own "computing home".

This guide will teach you how to do more than just hang pictures on a wall; it will show you how to build your dream home. While you read this user's guide, you will learn how to operate the MEGA65, write programs, add additional software, and extend hardware capabilities. What won't be immediately obvious is that along the journey, you will also learn about the history of computing as you explore the many facets of BASIC version 65 and operating system commands.

Computer graphics and music make computing more fun, and we designed the MEGA65 to be fun! In this user's guide, you will learn how to write programs using the MEGA65's built-in **graphics** and **sound** capabilities. But you don't need to be a programmer to have fun with the MEGA65. Because the MEGA65 includes a complete Commodore® 64™[2], it can also run thousands of existing games, utilities, and business software packages, as well as new programs being written today by Commodore computer enthusiasts. Excitement for the MEGA65 will grow as we all witness the programming marvels our MEGA65 community create, as they (and you!) discover and master the powerful capabilities of this modern Commodore computer recreation. Together, we can build a new "homebrew" community, teeming with software and projects that push the MEGA65's capabilities far beyond what anyone thought would be possible.

We welcome you on this journey! Thank you for becoming a part of the MEGA65 community of users, programmers, and enthusiasts!

[1] Commodore is a trademark of C= Holdings

[2] Commodore 64 is a trademark of C= Holdings

OTHER BOOKS IN THIS SERIES

This book is one of several within the MEGA65 documentation suite. The series includes:

- **The MEGA65 User's Guide**
  Provides an introduction to the MEGA65, and a condensed BASIC 65 command reference
- **The MEGA65 BASIC 65 Reference**
  Comprehensive documentation of all BASIC 65 commands, functions and operators
- **The MEGA65 Chipset Reference**
  Detailed documentation about the MEGA65 and C65's custom chips
- **The MEGA65 Developer's Guide**
  Information for developers who wish to write programs for the MEGA65
- **The MEGA65 Complete Compendium**
  (Also known as **The MEGA65 Book**)
  All volumes in a single huge PDF for easy searching. 1080 pages and growing!

COME JOIN US!

Get involved, learn more about your MEGA65, and join us online at:

- `https://mega65.org/chat`
- `https://mega65.org/forum`

CHAPTER 2

Setup

- Unpacking and Connecting the MEGA65
- Rear Connections
- Side Connections
- MEGA65 Screen and Peripherals
- Optional Connections
- Operation

UNPACKING AND CONNECTING THE MEGA65

Time to set up your MEGA65 home computer! The box contains the following:

- MEGA65 computer
- Power supply (black box with socket for mains supply)
- This book, the MEGA65 User's Guide

In addition, to be able to use your MEGA65 computer you may need:

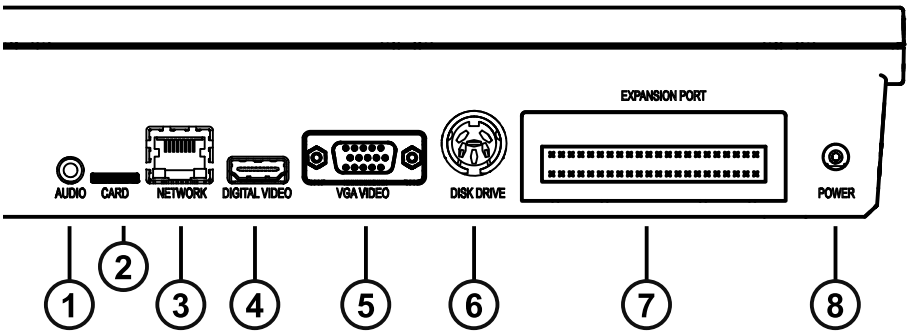
- A television or computer monitor with a VGA or digital video input, that is capable of displaying an image at 480p or 576p (720x480 or 720x576 pixel resolution at 50Hz or 60Hz)
- A VGA video cable, or;
- A digital video cable

These items are not included with the MEGA65.

You may also like to use the following to get the most out of your MEGA65:

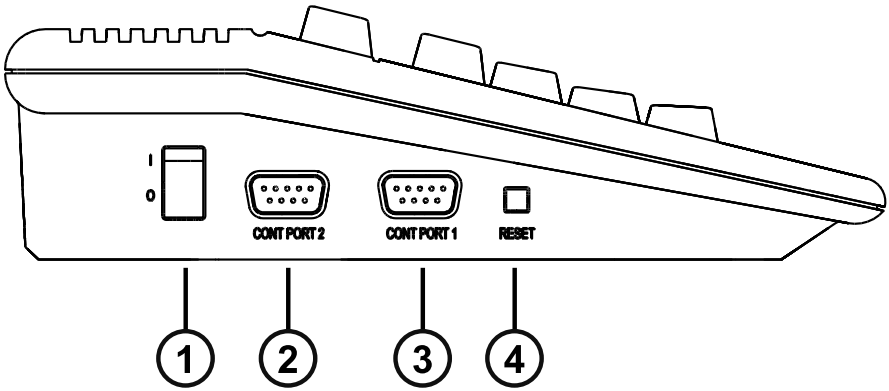
- 3.5mm mini-jack audio cable and suitable speakers or hi-fi system, so that you can enjoy the sound capabilities of your MEGA65.
- RJ45 Ethernet cable (regular network cable) and a network router or switch. This allows the usage of the high-speed networking capabilities of your MEGA65.

REAR CONNECTIONS



1	3.5mm Audio Mini-Jack
2	External microSD Card Slot
3	Network LAN Port
4	Digital Video Connector
5	VGA Video Connector
6	IEC Serial Bus Connector for Disk Drives and Printers
7	Cartridge Expansion Port
8	Power Supply Socket

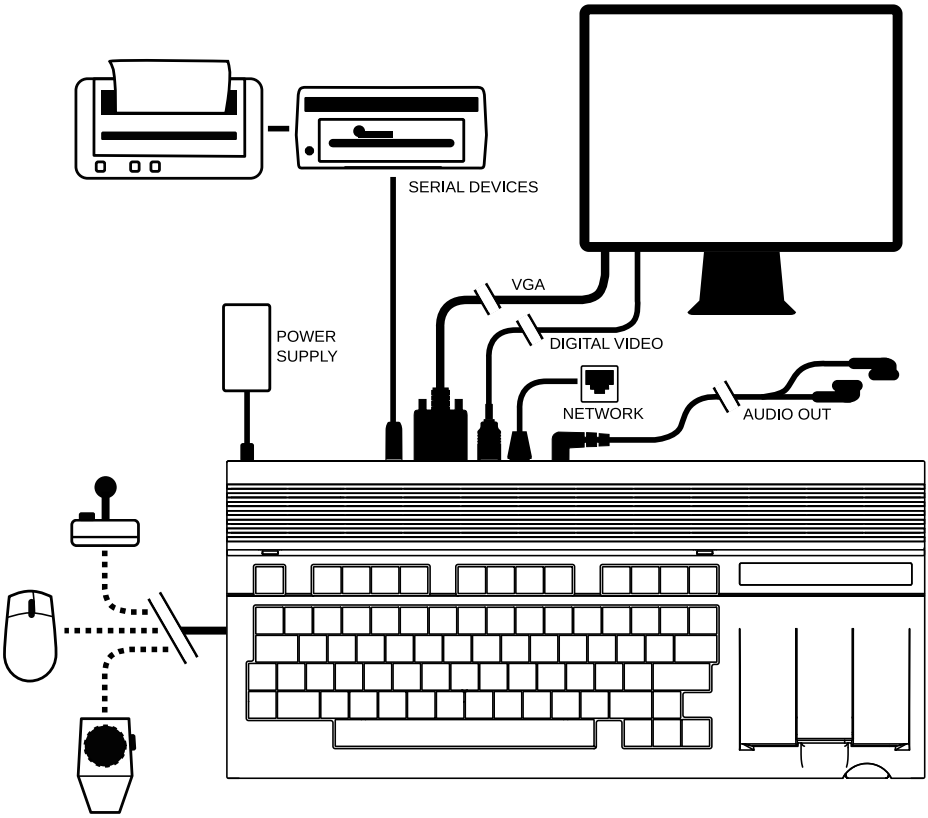
SIDE CONNECTIONS



1	Power Switch
2	Controller Port 2
3	Controller Port 1
4	Reset Button

Various peripherals can be connected to Controller Ports 1 and 2 such as joysticks, paddles or mouses.

MEGA65 SCREEN AND PERIPHERALS



1. Connect the power supply to the power supply socket of the MEGA65.
2. If you have a VGA monitor and a VGA cable, connect one end to the VGA port of the MEGA65 and the other end into your VGA monitor.
3. If you have a TV or monitor with a compatible Digital Video connector, connect one end of your cable to the Digital Video port of the MEGA65, and the other into the Digital Video port of your monitor. If you own a monitor with a DVI socket, you can use a Digital Video to DVI adapter.

OPTIONAL CONNECTIONS

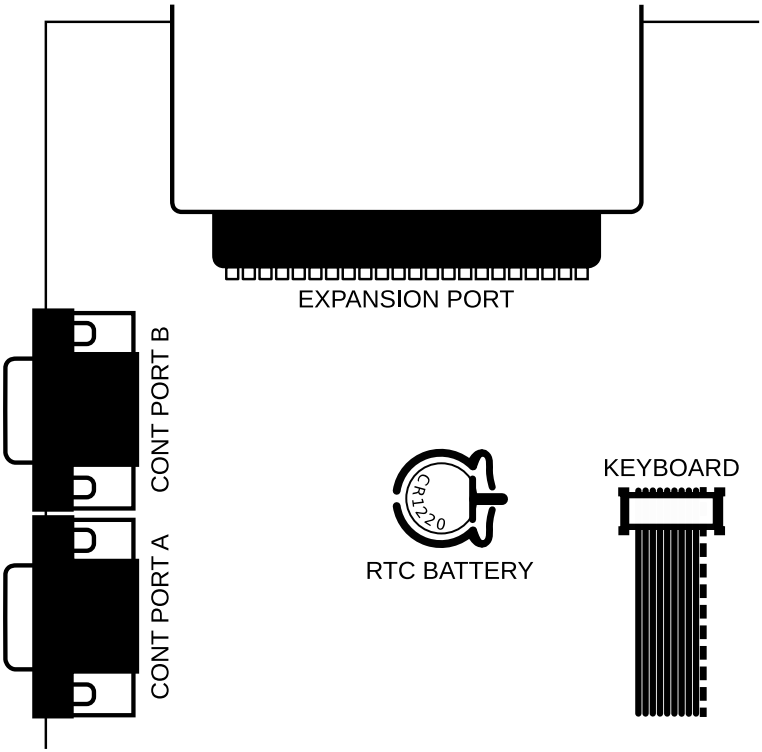
1. The MEGA65 includes an internal 3.5" floppy disk drive. You can also connect older Commodore® IEC serial floppy drives to the MEGA65, such as the Commodore 1541, 1571 or 1581. To use these drives, connect one end of an IEC cable to the Commodore floppy disk drive and the other end to the Disk Drive socket of the MEGA65. You can also connect SD2IEC devices and Pi1541's. It is also possible to daisy-chain additional floppy disk drives or Commodore compatible printers.
2. You can connect your MEGA65 to an Ethernet network using a standard Ethernet cable.
3. For enjoying audio from your MEGA65, you can connect a 3.5mm stereo mini-jack cable to an audio amplifier or speaker system. If your system has RCA connectors you will need a 3.5mm mini-jack to twin RCA adapter cable. The MEGA65 also has a built-in amplifier to allow the use of headphones.
4. A microSD card (either SDHC or SDXC) can be inserted into the external microSD card slot at the rear of the MEGA65.
5. Underneath the MEGA65, you will find an opening/trapdoor that provides access to the internal SD card slot, and also two PMOD connecters that allow for future possible hardware expansions such as Tape adapters, Userport interfaces, extra memory, or even real SIDs.

REPLACING THE REAL-TIME CLOCK BATTERY

The MEGA65 includes a Real-Time Clock, which is used to display the time and date on the startup screen, to add timestamps to files that the MEGA65 writes to your SD cards, and by the **DT$** and **TI$** BASIC65 commands. This clock utilises a CR1220 coin-cell battery to keep time when the MEGA65 isn't switched on, but unfortunately the MEGA65 doesn't ship with one (this is to minimise the chance of any international shipping related issues).

To change the battery, you will need to open the case, exposing the motherboard. The case is held together with three screws, all of which are along the bottom of the front side of the case. Once the three screws have been removed, **carefully** take the top half of the case off, and note the orientation of the keyboard connector, before disconnecting it.

The battery is located between the controller ports and the keyboard connector:



If you are removing an existing battery, push the battery release lever on the bottom (flat-sided) side of the battery socket *away* from the battery to remove it. Next, insert the new battery with the side labelled **+ facing up**, and press it into place.

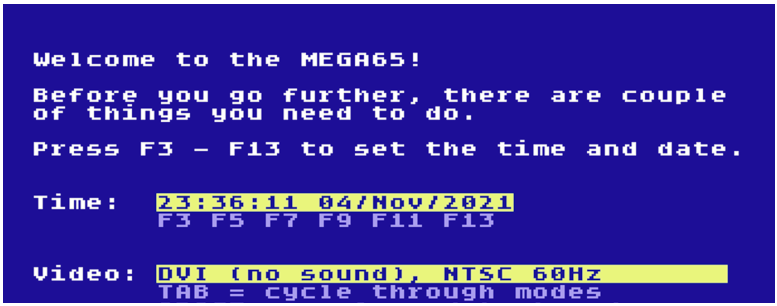
Once you have re-assembled your MEGA65, you can set the time in the Configure menu. For more information on how to set the Real-Time Clock, refer to the Configuration Utility section on page 31.

OPERATION

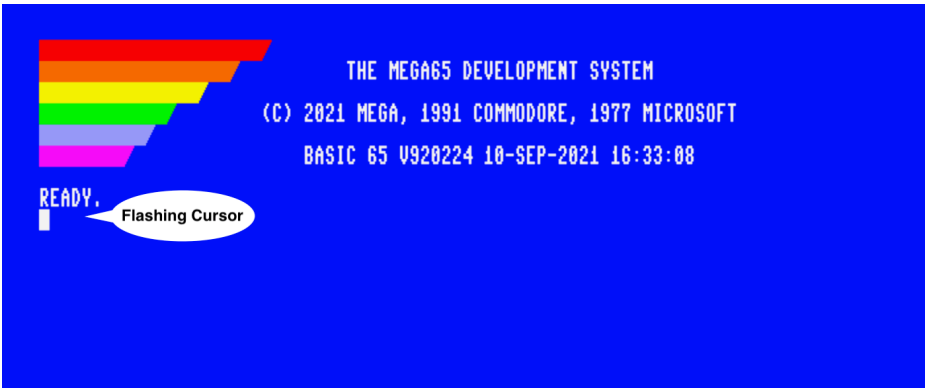
USING THE MEGA65

1. Switch on the MEGA65 by using the switch on the left-hand side.

   Note: On first power up, you will see an On-Boarding screen. See page 28 for more details.


Welcome to the MEGA65!
Before you go further, there are couple of things you need to do.
Press F3 - F13 to set the time and date.
Time: 23:36:11 04/Nov/2021
F3 F5 F7 F9 F11 F13
Video: DVI (no sound), NTSC 60Hz
TAB = cycle through modes

2. After a moment, the following will be displayed on your TV or monitor:



THE CURSOR

The flashing square underneath the READY prompt is called the cursor. The cursor indicates that the computer is ready to accept input. Pressing keys on the keyboard will print their respective characters onto the screen. The characters will be printed at the current cursor position, and the cursor will advance to the next position after every key press.

Here you can type commands, that can do things such as loading a program. You can also start entering program code!

CHAPTER 3

Getting Started

- Keyboard
- The Screen Editor
- Editor Functionality

KEYBOARD

Now that everything is connected, it's time to get familiar with the MEGA65 keyboard.

You may notice that the keyboard is a little different from the keyboards used on computers today. While most keys will be in familiar positions, there are some specialised keys, and some with special graphic symbols marked on the front.

Here's a brief description of how some of these special keys function.

COMMAND KEYS

The Command Keys are: RETURN, SHIFT, CTRL, , and RESTORE.

RETURN

Pressing RETURN enters the information you have typed into the MEGA65's memory. The computer will either act on a command, store some information, or display an error message if you made a mistake.

SHIFT

The two SHIFT keys are located on the left and the right. They work very much like the Shift key on a regular keyboard, however they also perform some special functions as well.

In upper case mode, holding down SHIFT and pressing any key with two graphic symbols on the front produces the right-hand symbol on that key. For example, SHIFT and J prints the ◹ character.

In lower case mode, pressing SHIFT and a letter key prints the upper case letter on that key.

Finally, holding SHIFT down and pressing a Function key accesses the function shown on the front of that key. For example: SHIFT and F1 activates F2.

SHIFT LOCK

In addition to SHIFT is SHIFT LOCK. Press this key to lock down the Shift function. Now any key you press while SHIFT LOCK is illuminated prints the character to the screen as if you were holding down SHIFT. This includes special graphic characters.

CTRL

CTRL is the Control key. Holding down CTRL and pressing another key allows you to perform Control Functions. For example, holding down CTRL and one of the number keys (from 1 to 8) allows you to change text colours. The colour that is printed at the top row on the front of the number key will be used. Holding down CTRL and pressing 9 or 0 switches reverse-text mode on and off.

There are some examples of this on page 15, and all of the Control Functions are listed on page 219.

If a program is being **LIST**ed to the screen, holding down CTRL slows down the display of each line. You can read more about the **LIST** command on page 146.

Holding CTRL and pressing * enters the Matrix Mode Debugger (refer to the **MEGA65 Book** for more details).

RUN STOP

Normally, pressing RUN STOP stops the execution of a program. Holding SHIFT while pressing RUN STOP **LOAD**s the first program from disk.

Programs are able to disable RUN STOP.

You can boot your MEGA65 into the **Machine Code Monitor** by holding down RUN STOP and pressing reset on the left-hand side.

RESTORE

The computer screen can be restored to a clean state without clearing the memory by holding down RUN STOP and pressing RESTORE. This combination also resets operating system vectors and re-initialises the screen editor, which makes it a handy combination if the computer has become a little confused.

Programs are able to disable this key combination.

You can also enter the **Freezer** by pressing and holding RESTORE for half to one second. From there you can access the Machine Code Monitor. You can read more about the Freezer on page 62.

THE CURSOR KEYS

At the bottom right-hand side of the keyboard are the cursor keys. These four directional keys allow you move the cursor to any position for on-screen editing.

The cursor moves in the direction indicated on the keys: ← ↑ → ↓.

However, it is also possible to move the cursor up by using SHIFT and ↓. In the same way you can move the cursor left by using SHIFT and →.

You don't have to keep pressing a cursor key over and over. If you need to move the cursor a long way, you can keep the key pressed down. When you are finished, simply release the key.

ARROW KEYS

These keys are different to the cursor keys! They are ← (next to 1), and ↑ (next to RESTORE). Both arrow keys are used in various BASIC functions and escape sequences.

For example, ← can be used as a shortcut for **SAVE**, and ↑ is used to raise a number to a power (which is the same as multiplying a number by itself a specified number of times).

You can read more about the available escape sequences on page 222.

These two PETSCII specific keys will always be shown in MEGA65 literature with a white background.

INSerT/DELete

This is the INSERT / DELETE key. When pressing INST DEL, the character to the left is deleted, and all characters to the right are shifted one position to the left.

To insert a character, hold SHIFT and press INST DEL. All the characters to the right of the cursor are shifted to the right. This allows you to type a letter, number or any other character at the newly inserted space.

CLeaR/HOME

Pressing CLR HOME places the cursor at the top left-most position of the screen.

Holding down SHIFT and pressing CLR HOME clears the entire screen *and* places the cursor at the top left-most position of the screen.

MEGA KEY

MEGA or the MEGA key provides a number of different functions and can be used to launch special utilities.

Holding SHIFT and pressing MEGA switches between lower and uppercase character modes.

Holding MEGA and pressing any key with two graphic symbols on the front prints the left-most graphic symbol to the screen. For example, MEGA and D prints the ◪ symbol.

Holding MEGA and pressing any key that shows a single graphic symbol on the front prints that graphic symbol to the screen.

Holding MEGA and pressing a number key switches to one of the colours in the second range, i.e., the colour that is printed at the bottom row on the front of the number key will be used.

Holding MEGA and pressing TAB enters the Matrix Mode Debugger (refer to the **MEGA65 Book** for more details).

Switching on the MEGA65 or pressing the reset button on the left-hand side while holding down MEGA switches the MEGA65 into C64-mode.

NO SCROLL

If a program is being **LIST**ed to the screen, pressing NO SCROLL freezes the screen output. This feature is not available in C64-mode.

FUNCTION KEYS

There are seven Function keys available for use by software applications, F1 F3 F5 F7 F9 F11 and F13 can be used to perform specific functions with a single press.

Hold SHIFT to access F2 through to F14 as shown on the front of each Function key. Only Function keys F1 to F8 are available in C64-mode.

HELP

HELP can be used by software and also acts as F15 / F16.

ALT

Holding ALT down while pressing other keys can be used by software to perform specific functions. Not available in C64-mode.

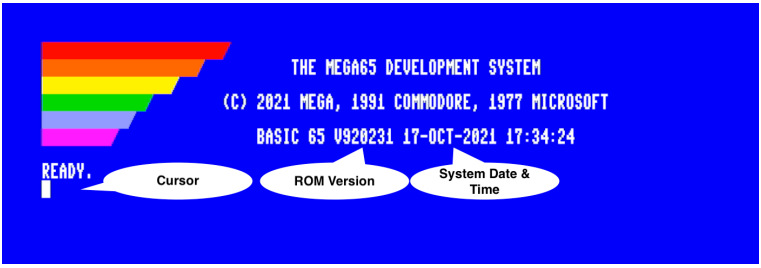
Holding ALT down while switching the MEGA65 on activates the Utility Menu. You can format an SD card, or enter the MEGA65 Configuration Utility to select the default video mode and change other settings, or to test your keyboard.

CAPS LOCK

CAPS LOCK works similarly to SHIFT LOCK in C65 and MEGA65-modes, but only modifies the letter keys. Also, holding CAPS LOCK down forces the processor to run at the maximum speed. This can be used for things such as speeding up loading from the internal disk drive or SD card, or to greatly speed up the de-packing process after a program is run. This can reduce the loading and de-packing time from many seconds to as little as a fraction of a second.

THE SCREEN EDITOR

When you switch on your MEGA65 or reset it, the following screen will appear:



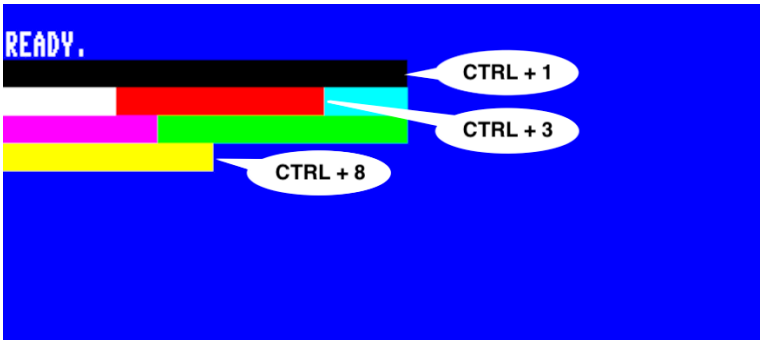
The colour bars in the top left-hand side of the screen can be used as a guide to help calibrate the colours of your display. The screen also displays the name of the system, the copyright notice, and the ROM version. The displayed date and time are taken from the internal RTC (Real-Time Clock), which can be set in the Configure Menu.

Finally, you will see the READY prompt and the flashing cursor.

You can begin typing keys on the keyboard and the characters will be printed at the cursor position. The cursor itself will advance after each key press.

You can also produce reverse text or colour bars by holding down CTRL and pressing 9, or R. This enters reverse text mode. When this is enabled, you can press

and hold the SPACE bar. While doing so, a white bar will be drawn across the screen. You can even change the current colour by holding CTRL down and pressing a number key (from 1 to 8). For example, if you press and hold CTRL down and press 1, the colour will change to black. Now, when you hold down the SPACE bar, a black bar will be drawn. If you continue to change the colour and press the SPACE bar, you will get an effect similar to the following image:



You can disable reverse text mode by holding CTRL and pressing 0.

By pressing any key, characters will be printed to the screen in the chosen colour.

A further eight colours can be selected by holding down C= and pressing a key from 1 to 8. The colour that is printed at the bottom row on the front of the number key will be used. For example, if you held C= down while pressing 4, dark grey will be used. For access to an additional 16 colours of the alternate/rainbow palette, refer to the CTRL + A shortcut described on page 219.

Note:

- **Quote Mode**: If you were to press " to open a string, and then try to change colours, reverse text, move the cursor keys, or use the CLR HOME key, instead of these actions instantly occurring, funny PETSCII symbols will appear instead. This is due to a BASIC facility called *quote mode* (described further in the **MEGA65 Book**), which allows you to encode such actions into a string so that they can be executed at a later time (for example, via a **PRINT** statement within your programs). To end *quote mode*, simply type another " to mark the end of your string.

- **Insert Mode**: A similar facility is called *insert mode*, where for the number of times you press SHIFT + INST DEL to insert a few spaces, the same number of keypresses that follow it will abide by the same principles of *quote mode*.
- You can forcefully exit either of these modes by pressing ESC, O.

You can create fun pictures just by using these colours and letters. Here's an example of what a year four student drew:

What will you draw?

Functions

Functions using CTRL are called **Control Codes**. Functions using C= are called **Mega Codes**. There are also functions that are called by using SHIFT, which are called **Shifted Codes**.

Lastly, ESC enables the use of **Escape Sequences**.

You can read about all of these functions in detail on page 219, but some are shown in this section.

ESC Sequences

Escape sequences are performed a little differently than a Control function or a Shift function. Instead of holding the modifier key down, an Escape sequence is performed by pressing ESC and releasing it, followed by pressing the desired key.

For example: to switch between 40/80 column mode, press and release ESC, then press X.

There are more modes available. You can create flashing text by holding CTRL down and pressing O. Any characters you type in will flash. Turn flash mode off by pressing ESC, then O.

EDITOR FUNCTIONALITY

The MEGA65 screen can allow you to do advanced tabbing, and quickly move around the screen in many ways to help you to be more productive.

For example, press CLR HOME to go to the home position on the screen. Hold CTRL down and press W several times. This is the **Word Advance function**, which jumps your cursor to the next word, or printable character.

You can set custom tab positions on the screen for your convenience. Press CLR HOME and then → to move the cursor to the fourth column. Hold down CTRL and press X to set a tab. Move another 16 positions to the right, and press CTRL and X again to set a second tab.

Press CLR HOME to go back to the home position. Hold CTRL down and press I. This is the **Forward Tab function**. Your cursor will tab to the fourth position. Press CTRL and I again. Your cursor will move to position 8. Why do you ask? By default, every 8th position is already set as a tabbed position. So the 4th and 20th positions have been added to the existing tab positions. You can continue to press CTRL and I to advance to the 16th and 20th positions.

Creating a Window

You can set a window on the MEGA65 working screen. Move your cursor to the beginning of the "BASIC 65" text. Press ESC, then press T. Move the cursor 10 lines down and 15 to the right.

Press ESC, then B. Anything you type will be contained within this window.

For example, if you were to type `LIST` to list out a program, the listing will be confined to the window region you have specified:

To escape from the window back to the full screen, press CLR HOME twice.

Additional ASCII characters

You may have noticed a few ASCII characters on the MEGA65 keyboard that aren't traditionally a part of the PETSCII character set. In order to make use of these from within BASIC:

- Type either `FONT A` or `FONT B`.
- Press ⌘ + SHIFT to switch to lowercase.

You will now be able to type those additional ASCII characters via the keyboard. To revert back to the original PETSCII character set, type `FONT C`.

Extras

Here are some extra, useful things you can do whilst in the screen editor:

To enter the **Freezer**, press and hold RESTORE for half to one second, then press J to switch controller ports without having to physically swap the controller to the other port. You can read more about the Freezer, and what else it can do on page 62.

To enter **Fast mode** (40.5MHz), either type `SPEED` (preferred way), or `POKE 0,65` or enter the Freeze Menu and switch the CPU frequency with F.

Return back to **Slow mode** (1MHz) by either typing `SPEED 1` (preferred way), or `POKE 0,64` or enter the Freeze Menu and switch the CPU frequency with F.

⌘ + SHIFT switches between uppercase and lowercase text for the entire display.

CHAPTER 4

Configuring your MEGA65

- Important Note
- Formatting SD cards
- Installing ROM and Other Support Files
- On-boarding
- Configuration Utility

IMPORTANT NOTE

For your convenience, your MEGA65 comes with an SD card with all of the essential files already on it, so you may prefer to skip this section and jump straight to the on-boarding section on page 28.

Alternatively, you're welcome to read this section and familiarise yourself on how your SD card was prepared.

Do not format the SD card that came with your MEGA65. If you want to create a new bootable SD card, please use another one, and keep the SD card that came with your MEGA65 as a safety backup.

FORMATTING SD CARDS

The MEGA65 has two SD card slots: A full-size SD card slot inside, under the trapdoor, and a microSD size slot on the rear. The current version of the MEGA65 firmware only supports the use of one SD card at a time. If you have cards in both slots, the MEGA65 will default to the external slot. The exception to this is that the MEGA65's FDISK/FORMAT utility can access both, allowing you to select which SD card to format or repair.

If you wish to use a different SD card to the pre-configured one supplied with your MEGA65, the new SD card must be formatted first.

This must be done on the MEGA65, not on a PC or other computer.

Only use SDHC cards. Older SD cards (typically with a capacity of <4GB) will not work. Newer SDXC cards with capacities greater than 32GB may or may not work. We would appreciate hearing your experience with such cards. It is unimportant as to which file-system is currently on the card, as the MEGA65 FDISK/FORMAT utility will completely reformat the card.

Why must I format the SD card via the MEGA65 (and not on a PC/computer)?

There are several reasons for this: First, to fit the most features into the MEGA65's small operating system, it is particular about the FAT32 file system it uses. Second, only the MEGA65 FDISK/FORMAT utility can create a MEGA65 System Partition. The MEGA65 System Partition holds non-volatile configuration settings for your MEGA65, and also contains the freeze slots that make it easy to switch between MEGA65 programs and games.

Formatting an SD card on the MEGA65 is easy. Switch the MEGA65 on while holding ALT down.

This will present the MEGA65 Utility Menu, which contains a selection of built-in utilities, similar to the following:

Note that the Utility Menu is always accessible, even if no SD card is present in both the internal and external slots.

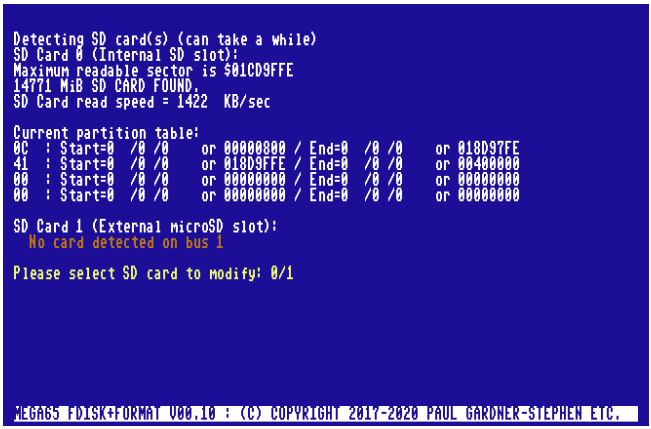
The exact set of utilities depends on the model of your MEGA65 and the version of the MEGA65 factory core which it is running. However, all versions include both the MEGA65 FDISK/FORMAT utility (which is called SDCARD FDISK+FORMAT UTILITY in the screenshot above), and the MEGA65 Configure utility. Most models also include a keyboard test utility, that you can use to test that your keyboard is functioning correctly. This is the same utility used in the factory when testing brand new keyboards.

Select the number that corresponds to the FDISK/FORMAT utility. This will typically be 2. The FDISK utility will start, and attempt to detect the size of all SD cards you have installed. If you have both an internal and external SD card installed, it will allow you to choose which one you wish to format. The internal SD card is bus 0, and the external microSD card is bus 1. Note that the MEGA65 will always attempt to boot from the external microSD card if one is present.

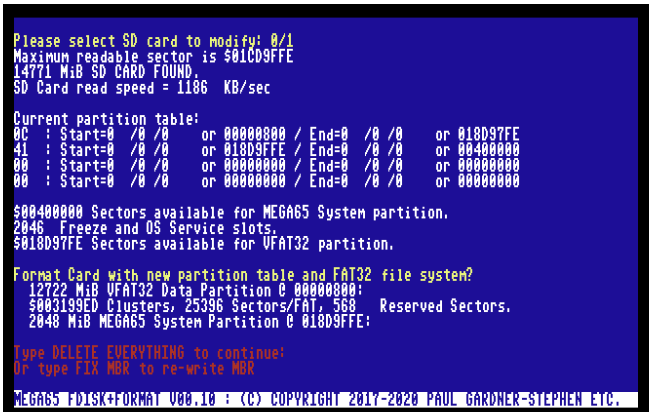
For safety, when formatting we *strongly* recommend that you remove any SD card or microSD card that you do not intend to format, so that you do not accidentally destroy any data. This is because formatting an SD card on the MEGA65 cannot be undone, and all data currently on the SD card *will be lost*. If you have files or data on the SD card that you wish to retain, you should first back them up. The contents of the FAT32 partition can be easily backed up by inserting the SD card into another computer. The contents of the MEGA65 System Partition, including the contents of freeze slots requires the use of specialised software.

You should aim to back up valuable data from your MEGA65 on a regular basis, especially while the computer remains under development. While we take every care to avoid data corruption or other mishaps, we cannot guarantee that the MEGA65 is free of bugs in this regard.

If you have only an internal SD card, you might see a display similar to the following:


Detecting SD card(s) (can take a while)
SD Card 0 (Internal SD slot):
Maximum readable sector is $01CD9FFE
14771 MiB SD CARD FOUND.
SD Card read speed = 1422 KB/sec
Current partition table:
0C : Start=0 /0 /0 or 00000800 / End=0 /0 /0 or 018D97FE
41 : Start=0 /0 /0 or 018D9FFE / End=0 /0 /0 or 00400000
00 : Start=0 /0 /0 or 00000000 / End=0 /0 /0 or 00000000
00 : Start=0 /0 /0 or 00000000 / End=0 /0 /0 or 00000000
SD Card 1 (External microSD slot):
No card detected on bus 1
Please select SD card to modify: 0/1
MEGA65 FDISK+FORMAT V00.10 : (C) COPYRIGHT 2017-2020 PAUL GARDNER-STEPHEN ETC.

Once you have selected the bus, the FDISK/FORMAT utility will ask you to confirm that you wish to delete everything:


Please select SD card to modify: 0/1
Maximum readable sector is $01CD9FFE
14771 MiB SD CARD FOUND.
SD Card read speed = 1186 KB/sec
Current partition table:
0C : Start=0 /0 /0 or 00000800 / End=0 /0 /0 or 018D97FE
41 : Start=0 /0 /0 or 018D9FFE / End=0 /0 /0 or 00400000
00 : Start=0 /0 /0 or 00000000 / End=0 /0 /0 or 00000000
00 : Start=0 /0 /0 or 00000000 / End=0 /0 /0 or 00000000
$00400000 Sectors available for MEGA65 System partition.
2046 Freeze and OS Service slots.
$018D97FE Sectors available for VFAT32 partition.
Format Card with new partition table and FAT32 file system?
12722 MiB VFAT32 Data Partition @ 00000800:
$003199ED Clusters, 25396 Sectors/FAT, 568 Reserved Sectors.
2048 MiB MEGA65 System Partition @ 018D9FFE:
Type DELETE EVERYTHING to continue:
Or type FIX MBR to re-write MBR
MEGA65 FDISK+FORMAT V00.10 : (C) COPYRIGHT 2017-2020 PAUL GARDNER-STEPHEN ETC.

To avoid accidental loss of data, you must type **DELETE EVERYTHING** in capitals and press RETURN. Alternatively, switch the MEGA65 off and on to abort this process without causing damage to your data.

It is also possible to attempt a recovery from a lost Master Boot Record error (also known as the Boot Sector), by typing **FIX MBR** instead.

The aim here is to have a correctly formatted SD card with all of the essential files stored on it so the MEGA65 can properly boot. When switching on, the MEGA65 will search for, and boot using these files:

- `FREEZER.M65` (Freeze Menu program)
- `AUDIOMIX.M65` (Freeze Menu audio mixer utility)
- `C64THUMB.M65` (C64 thumbnail image used in freezer)

- C65THUMB.M65 (C65 thumbnail image used in freezer)
- MEGA65.ROM (128KB ROM file)
- MEGA65.D81 (Disk image)

Straight out of the box, the MEGA65 will only have one SD card installed, accessible via the trap-door under the case. This SD card contains all of the essential files needed to properly boot up. If an external microSD card is also detected during boot up, the MEGA65 will give it higher priority, and will try to boot from it instead. This means that the external microSD card needs to have the essential files on it, otherwise the MEGA65 will not boot up properly and will fall back to loading the OpenROM, which does not support all MEGA65 features. In general, if your MEGA65 cannot boot properly and falls back to OpenROM, your boot-up screen will look similar to this:

INSTALLING ROM AND OTHER SUPPORT FILES

The MEGA65 FDISK/FORMAT utility will add a copy of the Open ROMs project's C64-compatible ROM to your SD card, and will name it MEGA65.ROM.

For MEGA65 owners, we have replaced this file with the latest ROM from the 'Closed ROMs' project. It provides many improvements over the original/incomplete C65 ROMs. It contains the operating system, BASIC 65, CBDOS and the machine language monitor BSMON. This ROM is developed especially for the MEGA65 and can be identified by the version number 92XXXX.

However, you may have other ROMs that you wish to use on your MEGA65. You can copy as many of these as you wish onto the SD card, just make sure that they have the .ROM file extension. The default ROM should be called "MEGA65.ROM". These files must be 128KB in size, and use the same internal format as the ROMs intended for the C65. This means that the C64-mode KERNAL must be placed at offset $E000, a C65-mode BASIC at $A000, and a suitable character set at $D000.

You can optionally name your alternate ROMs as `MEGA65x.ROM`, replacing `x` with a number from 0 to 9. This allows you to quickly boot-up to your alternate ROMs by holding down a number from **0** to **9** prior to switching on your MEGA65.

Other important support files include `FREEZER.M65` and `AUDIOMIX.M65`, which allow you to use the MEGA65's integrated Freezer. More details are provided in the 'Floppy Disks, D81 Images, and the Freezer' chapter on page 61.

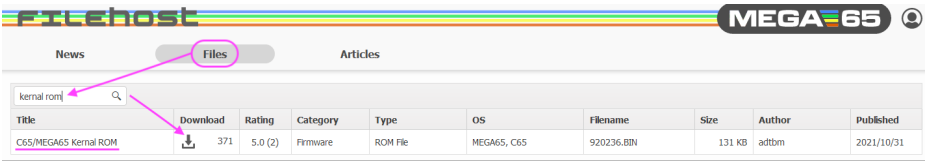
ROM FILE

Original C65 ROMs

You may want to source your own C65 ROM via other means. There were many different versions created during the development of the Commodore 65, and the MEGA65 can use any of them. However, they will not support the advanced features of the MEGA65, and are incomplete and buggy, as development on them ceased due to Commodore abandoning the C65 project.

MEGA65 Closed ROMs

There are newer versions of the MEGA65 Closed ROM under development. These ROMs improve upon the original C65 ROMs and make better use of the extra hardware capabilities that the MEGA65 has over the original C65 hardware. These ROMs are available via the filehost (`https://files.mega65.org`), but only to owners of the MEGA65, who will need to login to the filehost with their credentials in order to download it. It can be located by visiting the **Files** tab and searching for "**kernal rom**":



MEGA65 ROM diff files

If you have sourced your own `911001.bin` C65 ROM and would like to patch it to the latest MEGA65 closed ROM, we do provide patches, as the additional improvements we have made to the closed rom are open source. These diff files are available at `http://mega65.org/rom-diffs`

MEGA65 Open ROMs

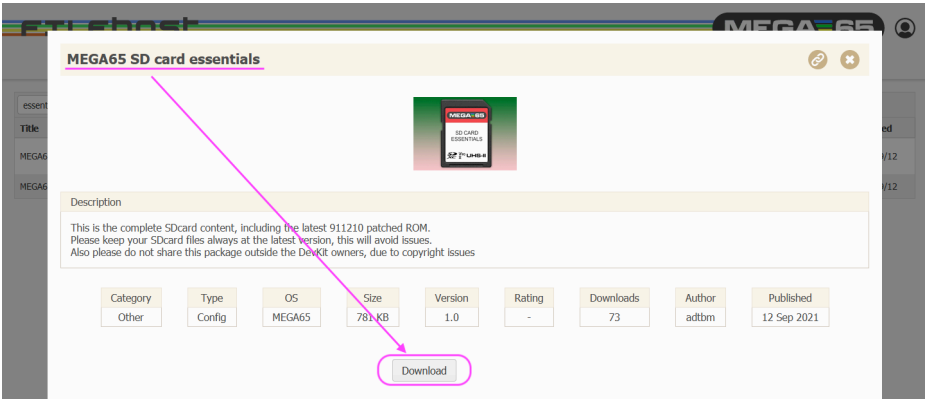
Another available option is to make use of **MEGA65 Open ROMs**. The latest version of this is always downloadable from either of the following urls:

- `http://mega65.org/open-roms`
- `https://github.com/MEGA65/open-roms/raw/master/bin/mega65.rom`

SUPPORT FILES

For official owners of the MEGA65 (both the devkit and the final product), visit the following url and log in with the user credentials you have been provided. This will take you to the MEGA65 filehost location where the **MEGA65 SD card essentials** download page is located. Click the **Download** link to retrieve the latest **SD essentials.rar** file.

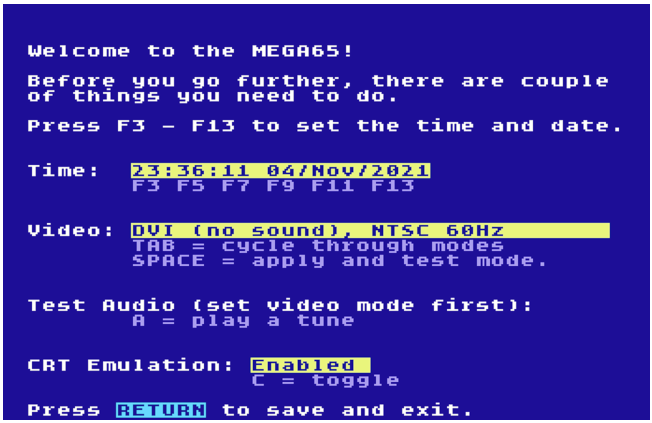
`http://mega65.org/sdcard-files`



Note that this link is only available to official owners of the MEGA65 product, as the fileset also contains the licensed closed-source `MEGA65.ROM` file.

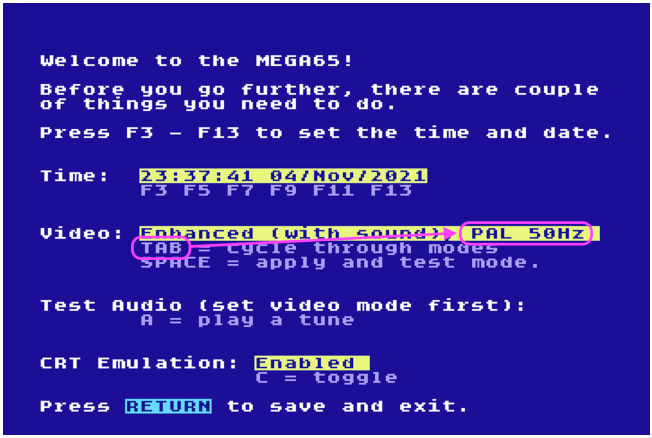
ON-BOARDING

On first launch of your MEGA65, you will see the on-boarding screen:

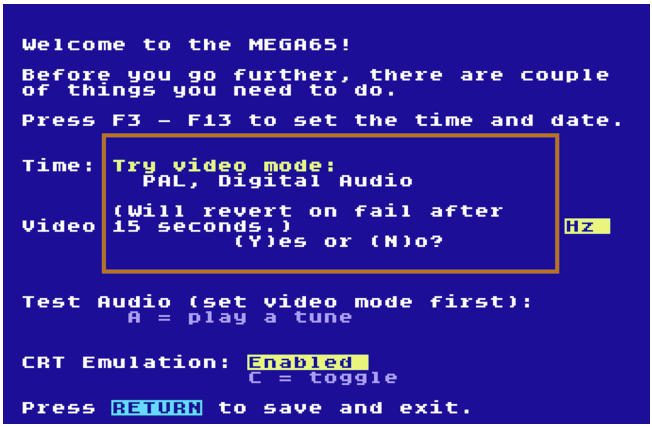

Welcome to the MEGA65!
Before you go further, there are couple of things you need to do.
Press F3 - F13 to set the time and date.
Time: 23:36:11 04/Nov/2021
F3 F5 F7 F9 F11 F13
Video: DVI (no sound), NTSC 60Hz
TAB = cycle through modes
SPACE = apply and test mode.
Test Audio (set video mode first):
A = play a tune
CRT Emulation: Enabled
C = toggle
Press RETURN to save and exit.

Here you can select and test you screen configuration.

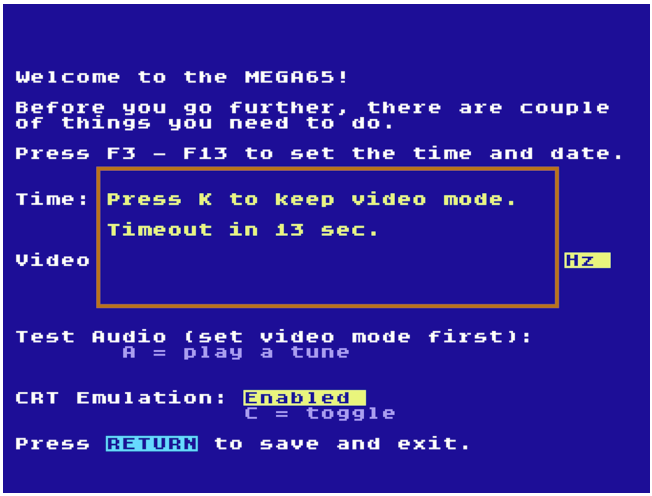
For example, press TAB to switch to PAL 50HZ:


Welcome to the MEGA65!
Before you go further, there are couple of things you need to do.
Press F3 - F13 to set the time and date.
Time: 23:37:41 04/Nov/2021
F3 F5 F7 F9 F11 F13
Video: Enhanced (with sound) PAL 50Hz
TAB = cycle through modes
SPACE = apply and test mode.
Test Audio (set video mode first):
A = play a tune
CRT Emulation: Enabled
C = toggle
Press RETURN to save and exit.

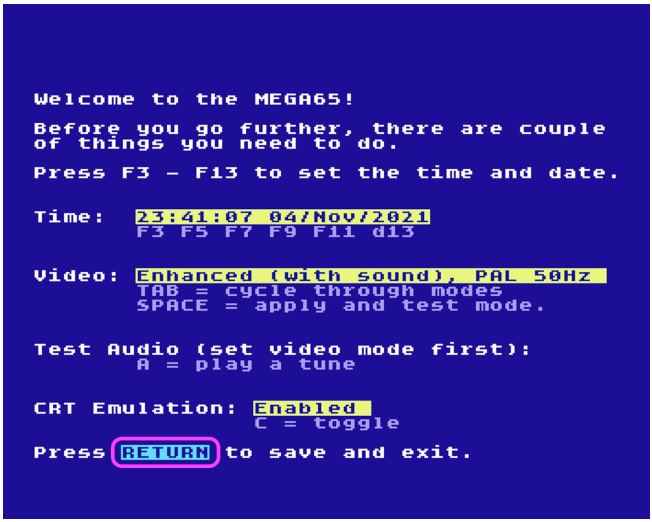
Then press SPACE, followed by Y to test the new video mode:


Welcome to the MEGA65!
Before you go further, there are couple of things you need to do.
Press F3 - F13 to set the time and date.
Time:
Try video mode:
PAL, Digital Audio
(Will revert on fail after 15 seconds.)
(Y)es or (N)o?
Video
Hz
Test Audio (set video mode first):
A = play a tune
CRT Emulation: Enabled
C = toggle
Press RETURN to save and exit.

Press K to keep the new video mode:


Welcome to the MEGA65!
Before you go further, there are couple of things you need to do.
Press F3 - F13 to set the time and date.
Time:
Press K to keep video mode.
Timeout in 13 sec.
Video
Hz
Test Audio (set video mode first):
A = play a tune
CRT Emulation: Enabled
C = toggle
Press RETURN to save and exit.

Press RETURN to complete the configuration:


Welcome to the MEGA65!
Before you go further, there are couple of things you need to do.
Press F3 - F13 to set the time and date.
Time: 23:41:07 04/Nov/2021
F3 F5 F7 F9 F11 d13
Video: Enhanced (with sound), PAL 50Hz
TAB = cycle through modes
SPACE = apply and test mode.
Test Audio (set video mode first):
A = play a tune
CRT Emulation: Enabled
C = toggle
Press RETURN to save and exit.

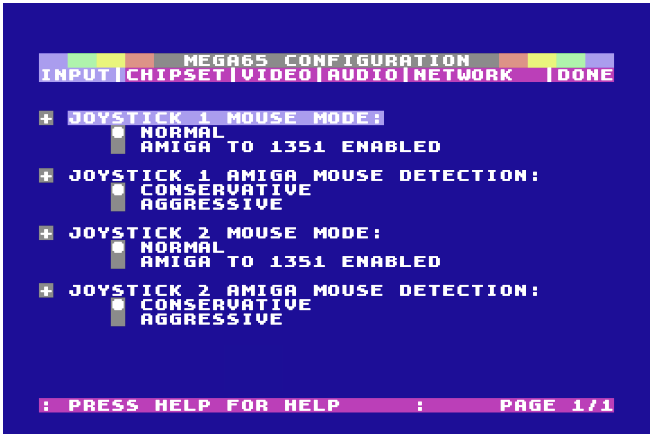
After the MEGA65 reboots you will be greeted with the main MEGA65 screen:

CONFIGURATION UTILITY

The configuration utility for the MEGA65 has a similar purpose to the BIOS on a PC, which is to allow you to control certain default behaviours of your MEGA65. However, rather than storing the configuration data in battery-backed RAM, the MEGA65 stores this data on sector 1 of the SD card. If you switch SD cards, you will change the configuration data.

To enter the configuration utility, switch the MEGA65 on while holding ALT down. This will show the utility menu, similar to the following:

Next, press the number corresponding to the CONFIGURE MEGA65 item. The MEGA65 Configuration Utility will launch, showing a display similar to the following:



If your MEGA65's System Partition is corrupted, you may be prompted to press F14 to correct this, (i.e., hold SHIFT and press F13), with a display similar to the following:

To correct this error, press F14. Next, press F7 to save the reset configuration, otherwise the reset data will not be saved to the MEGA65 System Partition.

Once you have dismissed that display, or if your MEGA65 System Partition is not corrupt, you can begin exploring and adjusting various settings. The program can be controlled using the keyboard, or optionally, a 1351 or Amiga™ mouse.

You can advance screens by pressing F1, or use F2 to navigate in the opposite direction. Use ← and → to navigate between screens.

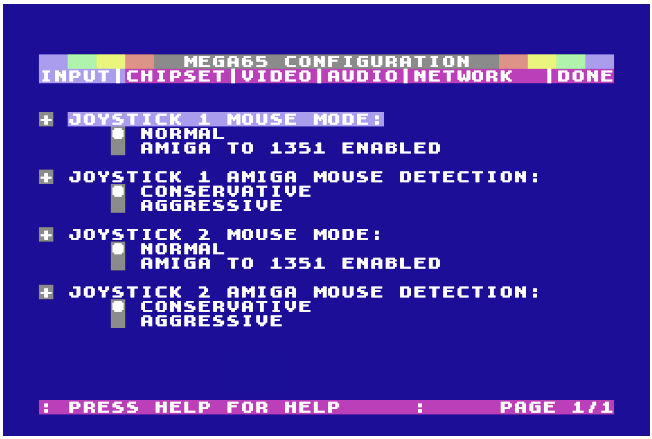
Use ↑ and ↓ to select an item.

Press RETURN or SPACE to toggle a setting, or to change a text or numeric value. The black circle next to an option indicates the current selection.

When you have finished, press F7 to see the options for saving the changes. This will give you four options:

- EXIT WITHOUT SAVING abandons any changes made in the MEGA65 Configure utility and exits.
- APPLY AND TEST SETTINGS NOW uses the current settings immediately but does not exit. This is helpful to test compatibility of your TV or monitor with PAL or NTSC video modes. If you still see your display after applying a change, it is safe to save those settings.
- RESTORE FACTORY DEFAULTS resets the MEGA65 configuration settings to the factory defaults. It will randomly select a new MAC address for models that include an internal Ethernet adaptor. If you wish to commit these changes, you must still save them.
- SAVE AS DEFAULTS AND EXIT commits changes made to the SD card. These changes will be used when the MEGA65 is switched on.

INPUT DEVICES

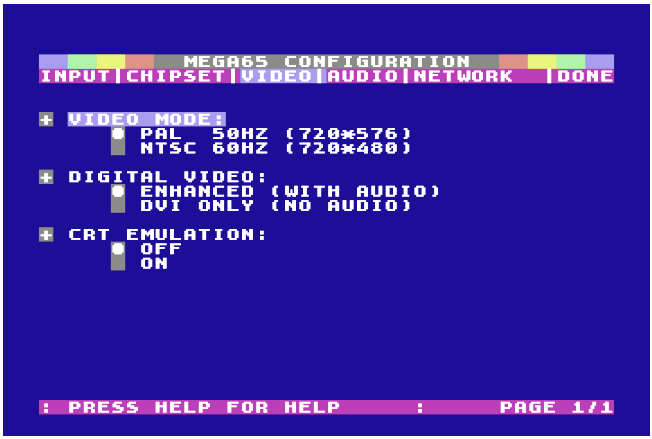


- JOYSTICK 1 AMIGA MOUSE MODE allows either NORMAL operation, where software will see it as an Amiga mouse or 1351 EMULATION mode, where the MEGA65 translates the Amiga mouse's movements into 1351 compatible signals. This allows you to use an Amiga mouse with existing C64/C65 software that expects a 1351 mouse.
- JOYSTICK 1 AMIGA MOUSE DETECTION can be set to CONSERVATIVE or AGGRESSIVE. If you use an Amiga mouse and it fails to move smoothly in all directions, you may set it to AGGRESSIVE. Conversely, if you regularly use joysticks in the port, and have difficulties with the joystick input misbehaving, you might want to use the CONSERVATIVE option.
- JOYSTICK 2 AMIGA MOUSE MODE is identical to the first option, but for the second controller port. This allows you to have different settings for each port.
- JOYSTICK 2 AMIGA MOUSE DETECTION similarly provides the ability to separately control the Amiga mouse detection algorithm for the second controller port.

CHIPSET

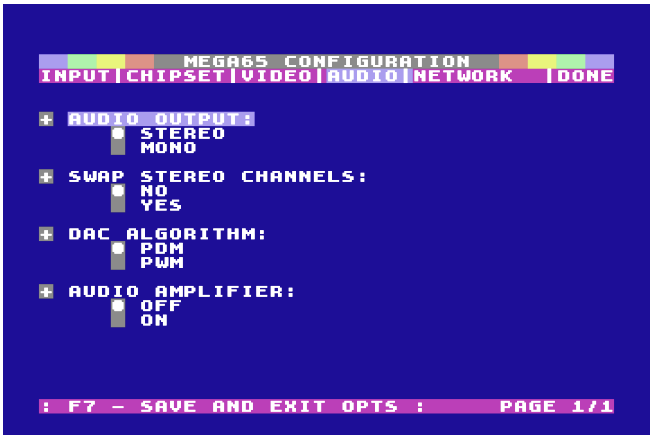
- REAL-TIME CLOCK allows setting the MEGA65's Real-Time Clock for those models that include one. To set the clock or calendar, simply edit the field and press RETURN. The display does not change while viewing this page, but if you use ← and → to select another page and return to this page, the values will update if a Real-Time Clock is fitted and functioning.
- DMAGIC REVISION allows selecting the default mode of operation for the C65 DMAgic DMA controller. This option is only required for ROMs not detected by the MEGA65's HYPPO Hypervisor. If you see screen corruption in BASIC, try toggling this option.
- F011 DISK CONTROLLER This option allows you to select whether the internal 3.5" floppy drive functions using real diskettes, or whether it simply makes noises to add atmosphere when using D81 disk images from the SD card. This merely sets the default option, and you can change this setting, or select a different disk image for use as either or both of the C65 3.5" DOS based drives.
- DEFAULT DISK IMAGE allows you to choose the D81 disk image used with the internal drive, if the F011 DISK CONTROLLER option above is set to USES SDCARD DISK IMAGE. You can read more about D81 disk images on page 64.
- LONG FN SUPPORT is a feature that is still under development at time of writing, and we suggest leaving it disabled for now until the feature matures in future bitstreams. Its aim is to provide long filename support for the SD card.

VIDEO



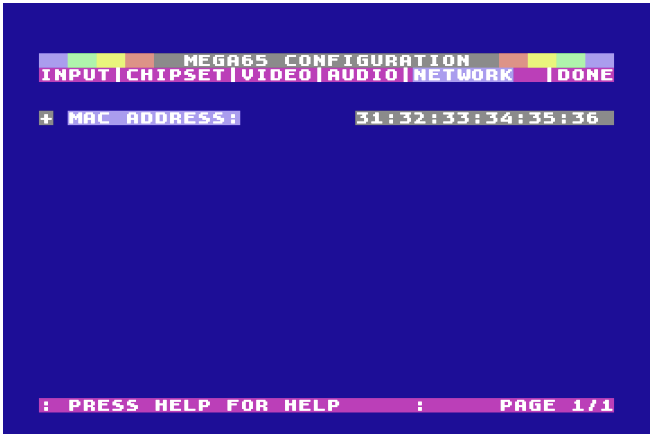
- VIDEO MODE selects whether the MEGA65 starts in PAL or NTSC. The MEGA65 supports true 480p NTSC and 576p PAL double-scan modes, with exact 60Hz / 50Hz frame-rates. This setting sets the default value, and the system can be switched between PAL and NTSC via the Freeze Menu, or under software control by MEGA65-enabled programs.
- DIGITAL VIDEO allows for selection between either ENHANCED video output containing audio, or DVI ONLY video output with no audio.
- CRT EMULATION selects whether CRT scanline emulation should be applied to the video output or not.

AUDIO



- AUDIO OUTPUT selects whether the SIDs and digital audio channels are combined to provide a monaural signal or whether the left and right tagged audio sources are separated to provide a stereo signal. This setting can be changed in the Audio Mixer of the Freeze Menu, or under the control of MEGA65-enabled software.
- SWAP STEREO CHANNELS allows switching the left and right-hand sides of the stereo audio output. This is useful for software that expects left and right SIDs to be at swapped addresses compared with the MEGA65 defaults.
- DAC ALGORITHM allows selecting between two different digital to analog conversion algorithms. Both options sound good and the selection is a personal preference.
- AUDIO AMPLIFIER allows enabling or disabling the audio amplifier contained in some models of the MEGA65. This option works for audio outputs, e.g., internal speaker or loud speaker.

NETWORK



- MAC ADDRESS allows you to set the default MAC address of your MEGA65. This can be changed at run-time by MEGA65-enabled software.

CHAPTER 5

C64, C65 and MEGA65 Modes

- Switching Modes from BASIC
- The KEY Register
- Accessing Memory from BASIC 65
- The MAP Instruction

The MEGA65, like the C65 and the C128, has multiple operating modes. However, there are important differences between the MEGA65 and both of these earlier computers.

By default, the `MEGA65.ROM` file boots to MEGA65-mode (including BASIC 65), and provides a method to switch to C64-mode via the `GO 64` command. However, it is also possible to use an original C65 ROM (version `91XXXX.BIN`) renamed to `MEGA65.ROM`, making the MEGA65 start in C65-mode with BASIC 10. This also provides the same functionality to switch to C64-mode.

Therefore, dependent on your boot ROM choice, you have:

Boot Mode	ROM version	BASIC	C64-mode
MEGA65	`92XXXX`	BASIC 65	`GO 64`
C65	`91XXXX`	BASIC 10	`GO 64`

For readers familiar with the C128, the most important difference is that all of the MEGA65's new features can be accessed from every mode, and that you can even switch back and forth between the different modes. It's also possible to create hybrid modes that combine different features from the different modes – all you need is the **MAP** instruction and the KEY register address, which is **53295** ($D02F).

This chapter explains the different modes, the **MAP** instruction, and the KEY register, which allows you to change the mode of operation of the MEGA65. This chapter also explains how to use BASIC commands to switch from one mode to another.

SWITCHING MODES FROM BASIC

The MEGA65 is used in either C64-mode (running BASIC 2), C65-mode (running BASIC 10) for ROM versions `91XXXX`, or MEGA65-mode (running BASIC 65) for ROM versions `92XXXX`.

However, various MEGA65 features can be accessed from all modes, and all MEGA65 features are available to programs written in assembly language / machine code. For more information on how to write such programs, refer to the **MEGA65 Book**.

FROM MEGA65/C65 TO C64-MODE

To switch from MEGA65/C65 to C64-mode, use the familiar `GO 64` command, which is identical to switching to C64 mode on a C128:

```
GO 64
ARE YOU SURE? Y
```

Note that any programs in memory will be lost in the process of switching modes. This is the same as the C128. Alternatively, you can hold down while pressing the reset button or switching the MEGA65 on. Again, this is the same as the C128.

FROM C64 TO MEGA65/C65-MODE

To switch from C64 to MEGA65/C65-mode, use the following command. **Note that this command does not ask you for confirmation**!

```
SYS 58552
```

Alternatively, you can switch back to MEGA65/C65-mode by pressing the reset button on the left-hand side of the MEGA65, or by switching the MEGA65 off and on again.

Another option is to press and hold RESTORE for between half to one second, and then press F5 from the Freeze Menu. This simulates pressing the reset button.

Note that any programs in memory will be lost in the process of switching modes. This is the same as the C128.

ENTERING MACHINE CODE MONITOR MODE

The Machine Code Monitor can be entered by typing either the **MONITOR** command from BASIC 65/10, or by holding RUN STOP down, and then pressing the reset button on the left-hand side of the MEGA65.

THE KEY REGISTER

The MEGA65 has a VIC-IV video controller chip instead of the C64's VIC-II or the C65's VIC-III. Just as the VIC-III has extra registers compared to the VIC-II, the VIC-IV has even more registers. If these were visible all the time, software that was made for the C64 and VIC-II may inadvertently use these new registers, resulting in unexpected behaviour. Therefore, the creators of the C65 created a way to hide the extra VIC-III registers from old C64 programs. Enabling and disabling (or hiding and un-hiding) the extra registers is done via the KEY register. For more information about which registers are disabled and enabled in each of the VIC-II, VIC-III and VIC-IV I/O modes, refer to the **MEGA65 Book**.

The KEY register, located at address **53295**, is an unused register of the VIC-II, which you can **POKE** to and **PEEK** from, similar to other registers. But the KEY register has a special function: If you write two certain values to it in quick succession,

you can tell the VIC-IV to stop hiding the VIC-III or VIC-IV registers from the rest of the MEGA65.

EXPOSING EXTRA C65 REGISTERS

For example, to enable the VIC-III's new registers when in C64-mode, you must **POKE** the values 165 and 150 into the KEY register. The easiest way to do this is to switch your MEGA65 off and on again, and type GO 64 and answer Y to enter C64-mode.

Note: If you perform these POKEs while in C65-mode, the MEGA65 may not function correctly.

Once you are in C64-mode, try typing the following commands:

```
POKE 53295,165: POKE 53295,150
```

When you enter these commands, the MEGA65 returns a **READY.** prompt, and seemingly nothing else has happened. This is expected, because the MEGA65 has only enabled the VIC-III's new registers (and some other C65-mode features). The C64 BASIC and KERNAL will still function as normal, and it may appear that nothing has changed... But things *have* changed.

For example, you can do something that the C64 and its VIC-II can't do: smoothly change one colour to another. The VIC-III has registers that allow you to change the red, green and blue components of the colours. Now that the VIC-III registers are enabled, it's possible to change the colour of the background progressively from blue to purple, by increasing the red component of the colour that is normally blue on the C64. The red component value registers are at 53504 - 53759 ($D100 - $D1FF). Blue is colour 6, so a change to register 53510 (53504 + 6, or $D106) is required. An example BASIC listing that includes a **FOR** loop to change the colour is:

```
FOR I = 0 TO 15 STEP 0.2 : POKE 53510,I : NEXT
```

Once the program has been entered, type **RUN** on a new line. This will make the background of the screen fade from blue to purple. If you would like to make the effect progress faster, increase the 0.2 to a larger number such as 0.5. To make it slower, change it to a smaller number such as 0.02. You can also change the red component by **POKE**ing a different number to 53504 - 53759 ($D100 - $D1FF), the green component at 53760 - 54015 ($D200 - $D2FF), or the blue component at 54016 - 54271 ($D300 - $D3FF). For example, to have the border and text (since they are both normally "light blue") fade from blue to green, you can try:

```
POKE 53518,0 : FOR I = 0 TO 15 STEP 0.1 : POKE 53774,I : POKE 54030,15-I : NEXT
```

DISABLING THE C65/MEGA65 EXTRA REGISTERS

You can also disable the VIC-III registers again by **POKE**ing any number into the KEY register, e.g.:

```
POKE 53295,0
```

If you **RUN** the examples above again, the colours won't change because the registers are disabled. Instead, writing to those addresses changes some of the VIC-II's registers, as on a C64 they appear several times over. Fortunately for the above example, the registers used have no obvious side-effects. This is because the modified registers in the examples above on a standard VIC-II are used to change the sprite positions. Since there are no sprites on the screen, you won't see anything change.

ENABLING MEGA65 EXTRA REGISTERS

The MEGA65 has *even more* registers than the C65. To enable these in C64-mode, it's required to **POKE** another two values into the KEY register:

```
POKE 53295,71: POKE 53295,83
```

Again, you won't see any immediate difference, which is similar to when enabling the VIC-III registers. However, now the MEGA65 can access not only the VIC-II and VIC-III registers, but also the VIC-IV registers. If you like, you can try the examples from earlier in this chapter to see that the VIC-III registers are accessible again. But now you can also do MEGA65 specific things. For example, if you wanted to move the start of the top border higher on the screen, you can try something such as:

```
POKE 53320,60
```

Alternatively, you can have some fun and animate the screen borders, by having them move closer and further apart:

```
FOR I = 255 TO 0 STEP -1 : POKE 53320,I : POKE 53322, 255 - I : NEXT
```

The above example has the loop count backwards (from 255 to 0), so that your don't end up with only a tiny sliver of the text visible. You can make it go forwards if you like. If you do get stuck with only a sliver of the screen, you can press RUN STOP and RESTORE. You might be wondering: Why does RUN STOP and RESTORE work when these are VIC-IV registers that the C64-mode BASIC and KERNAL don't know about? The

reason is the VIC-IV has a feature called "hot registers", where certain C64 and C65 registers cause some MEGA65 registers to be reset to the C64 or C65-mode defaults. In this particular case, it is the KERNAL resetting the VIC-II screen using 53265 ($D011), which adjusts the vertical border size in C64/C65-mode, and is thus a "hot register" for the MEGA65's vertical border position registers.

See if you can make the screen shake around instead by changing the TEXTXPOS and TEXTYPOS registers of the VIC-IV. You can find out the **POKE** codes for those, and lots of other interesting new registers by looking through the **MEGA65 Book**.

TRAPS TO LOOK OUT FOR

In all modes, the DOS for the internal 3.5" disk drive (including when you use D81 disk images from an SD card) resets the KEY register to VIC-II mode whenever it is accessed. This means if you perform actions such as check the drive status, or **LOAD** or **SAVE** a file, the KEY register will be reset, and only the VIC-II registers will be enabled. You can of course enable the C65 or MEGA65 registers by **POKE**ing the correct values to the KEY register again.

ACCESSING MEMORY FROM BASIC 65

BASIC 65 contains powerful memory banking and Direct Memory Access (DMA) commands that can be used to read, fill, copy, and write areas of memory beyond the C65's 128KB of RAM. The MEGA65 has 384KB of main memory, split into 6 banks of 64KB each. They are:

- BANK 0 and BANK 1 - acts as the C65's normal 128KB RAM.
- BANK 2 and BANK 3 - normally write-protected, and contains the C65's ROM image.
- BANK 4 and BANK 5 - used for all graphic routines in BASIC 65 for high resolution bitplane graphics. BASIC 10 doesn't use banks 4 and 5.

Using the **BANK**, **PEEK** and **POKE** commands, this region of memory can be easily accessed, for example:

```
BANK 4: POKE0,123: REM PUT 123 IN LOCATION $40000
BANK 4: PRINT PEEK(0): REM SHOW CONTENTS OF LOCATION $40000
```

Or, by using the **DMA** command, you can copy the current contents of the screen and colour RAM into BANK 4 with:

```
DMA 0, 2000, 2048, 0, 0, 4 : REM SCREEN TEXT TO BANK 4
DMA 0, 2000, DEC("F800"), 1, 2000, 4 : REM COPY COLOUR RAM TO BANK 4
```

You can then put something else on the screen, and copy it back with:

```
DMA 0, 2000, 0, 4, 2048, 0 : REM SCREEN TEXT FROM BANK 4
DMA 0, 2000, 2000, 4, DEC("F800"), 1 : REM COPY COLOUR RAM FROM BANK 4
```

THE MAP INSTRUCTION

The above methods can be used from BASIC. In contrast, the **MAP** instruction is an assembly language instruction that can be used to rearrange the memory that the MEGA65 uses. It is used by the C65 ROM and BASIC 65 to manage what memory it can use at any particular point in time. For further explanation of the **MAP** instruction, refer to the relevant section of the **MEGA65 Book**.

CHAPTER 6

Cores and Flashing

- What are Cores, and Why Do They Matter?
- Bitstream files
- Selecting a Core
- Installing an Upgrade Core for the MEGA65
- Installing Other Cores
- Creating Cores for the MEGA65
- Replacing the Factory Core in Slot 0
- Understanding The Core Booting Process
- DIP Switches

WHAT ARE CORES, AND WHY DO THEY MATTER?

The MEGA65 computer uses a versatile chip called an FPGA as its heart, which is an acronym for "Field Programmable Gate Array". This is a fancy way of saying that FPGAs are chips that can be programmed *by you* to impersonate other chips. They do this by re-configuring their arrays of logic gates to reproduce the circuits of other chips. As a result, FPGAs are not an emulation, but a re-creation of other chips.

However, FPGAs forget what chip they are pretending to be whenever the power is turned off, or when they are re-programmed. This might sound annoying, but it's actually very powerful. It means that you can tell the FPGA in the MEGA65 to impersonate not just the MEGA65 design as it currently stands, but to impersonate any improvements made to the design itself. In other words, you can upgrade the MEGA65 hardware just by providing a new set of instructions to the FPGA. These sets of instructions are called "cores", or "bitstreams". For the purpose of the MEGA65, these two terms are interchangeable.

FPGAs are so flexible that not only is it possible to teach the MEGA65 to be a better MEGA65, but it is also possible to teach the MEGA65 to be other interesting home computers. We believe that the FPGA is powerful enough to re-create a Commodore PET™, VIC-20™, Apple II™, Spectrum™, BBC Micro™, or even an Amiga™, or one of the 16-bit era game consoles. Unlike some previous FPGA-based retro-computers, the MEGA65, its FPGA instructions, board layout, and other information is all available for free under various open-source licenses. This means that anyone is free to create other cores for the MEGA65 hardware.

To top it all off, the MEGA65 has enough storage for 7 different sets of FPGA instructions, so that you can easily switch the MEGA65's "personality" from being a MEGA65 to another system, and back again.

The remainder of this chapter describes how to select a core to run on the MEGA65, and how to store a core into one of the seven slots in the flash memory storage.

BITSTREAM FILES

Firstly, there are a variety of files related to the MEGA65's cores/bitstreams that you should be familiar with, in order to decide what file-types are needed for what occasion.

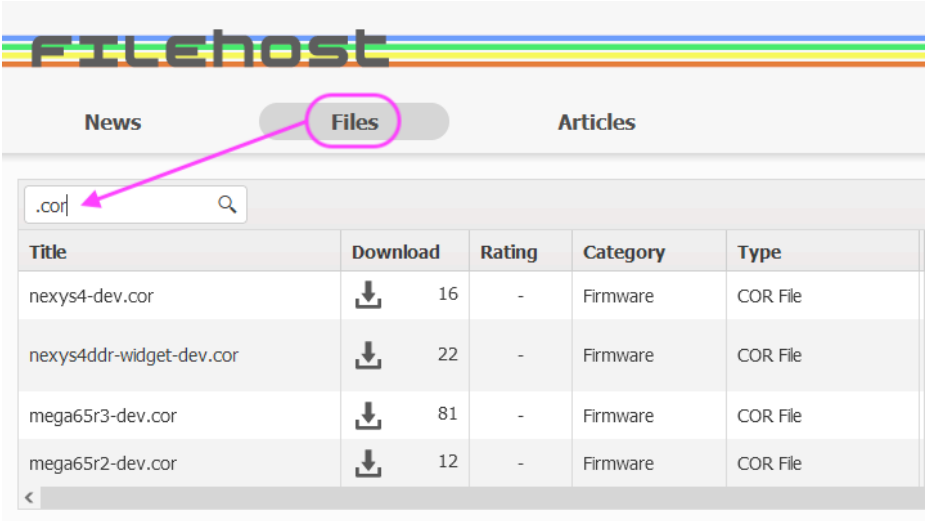
FILE TYPES

File-type	Purpose
`.cor`	The MEGA65 project's custom bitstream file format, containing extra header information to help identify the bitstream and the specific MEGA65 target device it is intended for. The MEGA65's flashing utility makes use of this additional information to ensure you don't accidentally flash the bitstream of a different device.
`.mcs`	The bitstream file in a format needed when flashing it to your device's QSPI flash memory chip via Vivado®.
`.prm`	This file contains checksum information that can be used by Vivado to verify the `.mcs` file you have tried to flash. Optional.
`.bit`	A plain bitstream file in its purest binary form.

WHERE TO DOWNLOAD

Visit the following url:

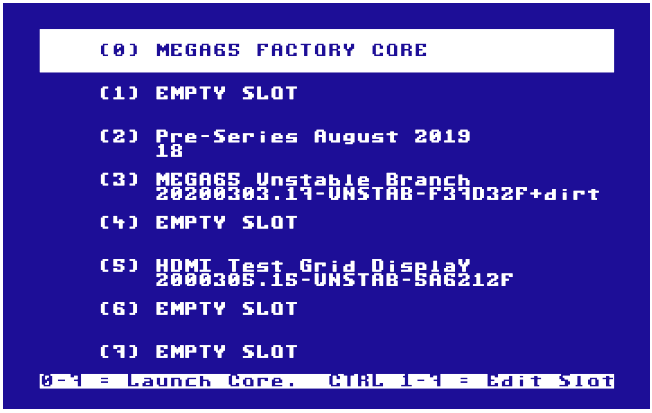
`https://files.mega65.org`



Click the **Files** tab, and in the search-bar, type **.cor** and press Enter. You will notice that there are different files with the `.cor` extension. For your MEGA65, download the file that ends with **65r3-dev.cor**. Other files are for other device types, which you can read more about in the **MEGA65 Book**.

SELECTING A CORE

To operate the MEGA65 with an alternate core, switch off the power to the MEGA65, and then hold NO SCROLL down while switching the power back on. This instructs the MEGA65 to enter the Flash and Core Menu, instead of booting normally. When booting this way, the following screen will appear:


(0) MEGA65 FACTORY CORE
(1) EMPTY SLOT
(2) Pre-Series August 2019 18
(3) MEGA65 Unstable Branch 20200303.14-UNSTAB-F39D32F+dirt
(4) EMPTY SLOT
(5) HDMI Test Grid Display 2000305.15-UNSTAB-5A6212F
(6) EMPTY SLOT
(7) EMPTY SLOT
0-7 = Launch Core. CTRL 1-7 = Edit Slot

To select a core and start it, use the cursor keys to highlight the desired core, and then press RETURN. If you select a flash slot that does not contain a valid core, it will be highlighted in red to indicate that it cannot be booted from:

(0) MEGA65 FACTORY CORE
(1) EMPTY SLOT
(2) Pre-Series August 2019 18
(3) MEGA65 Unstable Branch 20200303.14-UNSTAB-F39D32F+dirt
(4) EMPTY SLOT
(5) HDMI Test Grid Display 2000305.15-UNSTAB-5A6212F
(6) EMPTY SLOT
(7) EMPTY SLOT
0-7 = Launch Core. CTRL 1-7 = Edit Slot

Alternatively, you can press the number corresponding to the core you would like to use. The MEGA65 immediately reconfigures the FPGA, and launches the core. If for some reason the core is faulty, the MEGA65 may instead restart normally after a few seconds, and depending on the circumstances, take you back into the menu automatically.

The MEGA65 will keep running the new core until you physically power it off. Pressing the reset button will not reset which core is being run.

INSTALLING AN UPGRADE CORE FOR THE MEGA65

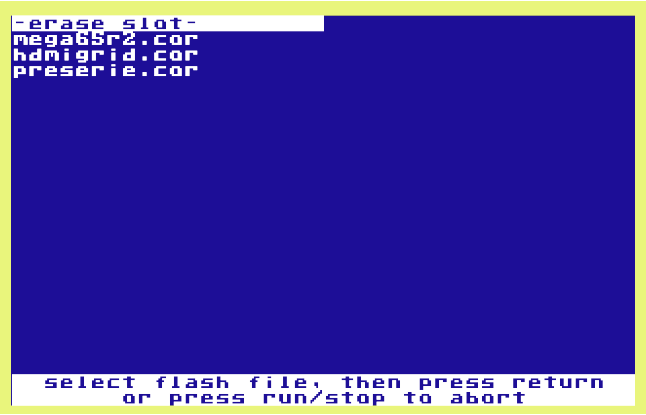
Installing and upgrading the core (from a `.cor` file) for the MEGA65 can be done in a few easy steps.

First, copy the core into the MEGA65 SD card's root directory. You can do this by removing the SD card and copying a previously downloaded core file to it from another computer. Alternatively, you can insert an SD card that already contains the upgrade core. Finally, you can use the MEGA65 TFTP Server program and the MEGA65's Ethernet port to upload the core upgrade file onto the SD card from another computer on your local network.

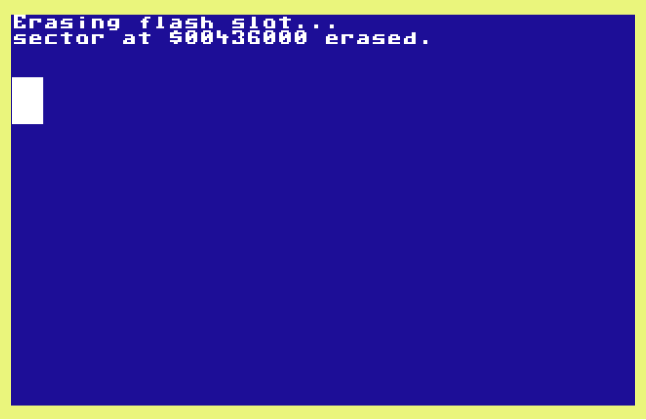
The Flash Menu will use the external microSD slot over the internal SD card, so if you have both a microSD card and SD card inserted in your MEGA65, the Flash Menu will ignore the internal SD card. To avoid this, simply copy the core(s) from the internal SD card to the external microSD card, or temporarily remove the external microSD card from the rear of your MEGA65, so that the Flash Menu will be able to find the core files. Also note that the Flash Menu currently only supports DOS-style 8.3 character filenames in UPPERCASE. If your core files have a longer name, you will need to rename them when copying them onto your microSD or SD card.

Note: If you plan on using your external microSD card solely for upgrading cores (and not for booting from), then you will not need to add "SD card essentials" files onto your microSD card (simply adding the .cor file will suffice).

Next, once you have the upgrade core on the MEGA65's SD card, enter the Flash and Core Menu as above, i.e., switch off the power, and hold NO SCROLL down while switching the power on again. When the Flash and Core Menu appears, hold CTRL down and press 1 (or CTRL and a different number, if you wish to replace the contents of a different flash slot). The MEGA65 will present you with a list of core files that are on the SD card:



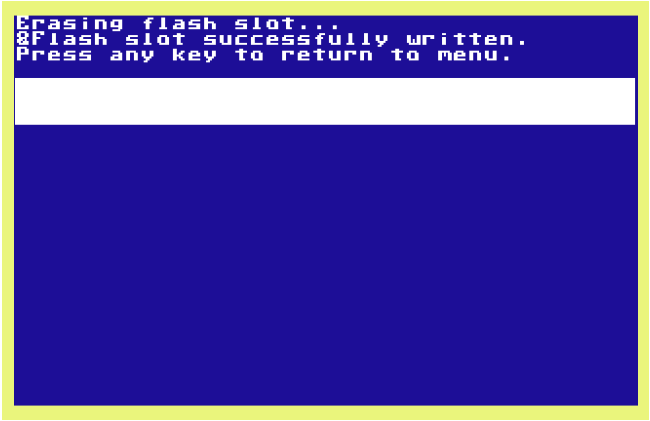
Select the upgrade core file you wish to install using the cursor keys, and then press RETURN. The MEGA65 will then erase the flash slot, before writing the upgraded core. You will see a progress bar while the MEGA65 erases the flash slot:



The progress bar will then reset, and the MEGA65 will write the new core into the slot. This process can take up to 15 minutes, depending on the size of the core file. If you simply wish to erase a flash slot, you can select the -- erase slot -- option instead of a file name. This will then perform only the erasure part of the process.

It is important to not switch the power off during this process. If you do, the core file will be only partially installed, and the MEGA65 may not start properly. While inconvenient, it won't damage your MEGA65 or leave it in an unusable state: It will simply fall back to using the factory supplied core. If this happens, enter the Flash and Core Menu as described above, and follow the instructions again.

When the flashing process has completed, you will see a message indicating that the process is complete:



When this happens, simply switch off the power to the MEGA65 and switch it back on again for it to start using the upgraded core. This is because the MEGA65 will always try to start the core in slot 1 when it is switched on.

Note: After flashing completes, it is safe to delete .cor files from your SD card, as their contents now reside within the QSPI flash memory.

INSTALLING OTHER CORES

Installing other cores works very similarly to installing upgrade cores. The only difference is that you press CTRL and 2 to 7 from the Flash and Core Menu, so that the core gets installed to another slot.

Of course, there is nothing stopping you from installing a different core in slot 1, so that the MEGA65 behaves as a different type of computer when you switch it on. If you do this, you can always choose to run the MEGA65 core by entering the Flash and Core Menu, and selecting the MEGA65 core.

CREATING CORES FOR THE MEGA65

If you would like to create your own cores for the MEGA65, or help contribute to the MEGA65 core, then you may also wish to take a look at the **MEGA65 Book**, which explains how to use the FPGA development tools to flash the MEGA65.

REPLACING THE FACTORY CORE IN SLOT 0

Replacing the core in slot 0 is not recommended, because if it ever gets corrupted, it will "brick" the machine. This will require you to connect a TE-0790 JTAG programmer, by opening your MEGA65 case, installing the module, going through some rather convoluted software preparation steps (similar to if you were creating your own bistream/core) and then restoring a working bitstream into the slot.

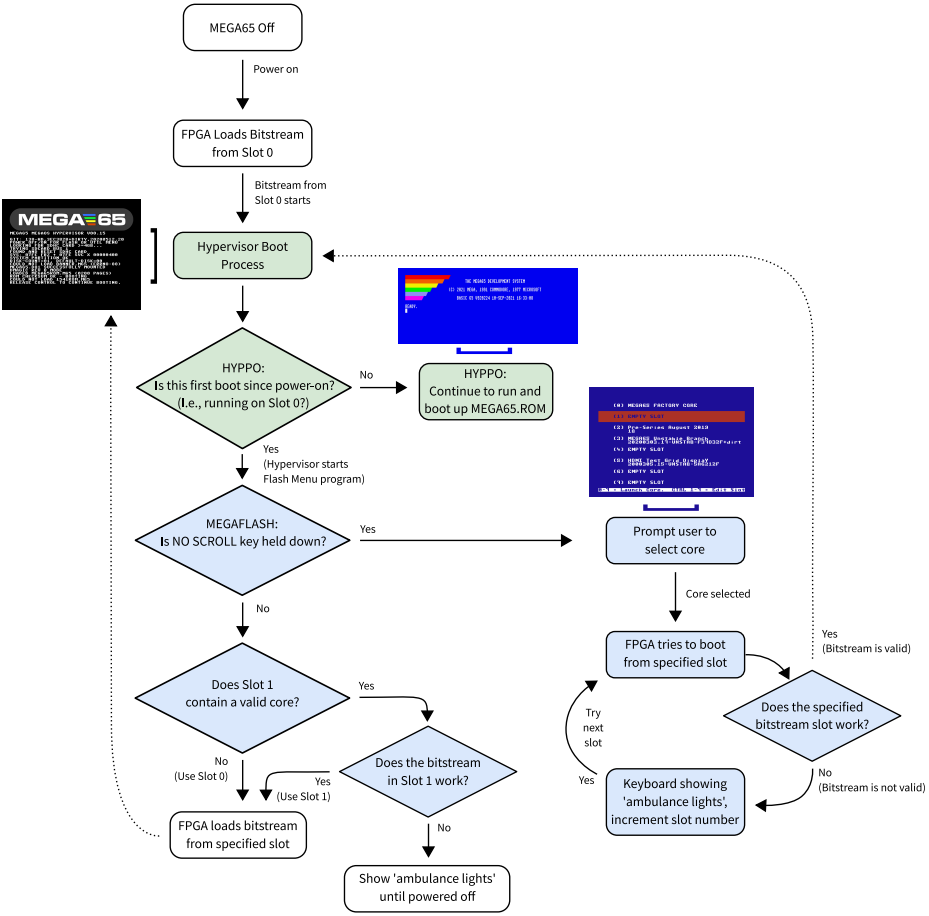
The MEGA65 is an open system though, so it's possible for you to do all of this, but it's very hard. There is a secret key-press combination in the Flash Menu that will then challenge you with a series of questions with increasing difficulty to ensure that you know what you are doing. Only after you have correctly answered these questions will you be given the option to erase and/or replace the contents of slot 0. Details of the questions asked are purposely not documented.

There really should be no reason for using this method to replace the contents of slot 0: If you want to make your own bitstreams/cores, you can either write them to other slots and use the Flash Menu to activate them, or you can simply use a TE-0790 JTAG programmer, and then use Vivado or other FPGA development tool to write to the flash directly. This method is also somewhat faster than flashing through the Flash Menu.

You have been warned!

UNDERSTANDING THE CORE BOOTING PROCESS

This section summarises how the MEGA65 selects which core to start with when it is switched on. The process is shown in the following figure:


MEGA65 Off
Power on
FPGA Loads Bitstream from Slot 0
Bitstream from Slot 0 starts
Hypervisor Boot Process
HYPPO: Is this first boot since power-on? (I.e., running on Slot 0?)
No
HYPPO: Continue to run and boot up MEGA65.ROM
Yes (Hypervisor starts Flash Menu program)
MEGAFLASH: Is NO SCROLL key held down?
Yes
Prompt user to select core
Core selected
No
Does Slot 1 contain a valid core?
Yes
Does the bitstream in Slot 1 work?
No (Use Slot 0)
Yes (Use Slot 1)
FPGA loads bitstream from specified slot
No
Show 'ambulance lights' until powered off
FPGA tries to boot from specified slot
Yes (Bitstream is valid)
Does the specified bitstream slot work?
No (Bitstream is not valid)
Keyboard showing 'ambulance lights', increment slot number
Yes
Try next slot

The booting process is governed by two facilities:

- The Hypervisor (also known as HYPPO), which operates at a level above the Kernal. One of its responsibilities is to manage aspects of the boot process. For more details on the Hypervisor, refer to the **MEGA65 Book**. In the diagram, activities performed by the Hypervisor have been highlighted in green.
- The Flash Menu program (also known as MegaFlash), which provides a list of available core slots to choose from. In the diagram, activities performed by MegaFlash have been highlighted in blue.

When the MEGA65 is switched on, it does the following:

- Loads the bitstream stored in slot 0 of flash memory. If that is the MEGA65 Factory Core, the MEGA65 HYPPO Hypervisor starts.

- If it is the first boot since power-on (which implies that we are running from slot 0), HYPPO starts the Flash Menu program (aka MegaFlash) - but note that the Flash Menu in this mode may not show anything on the screen to indicate that it is running!
- The Flash Menu then checks if NO SCROLL is being held down.
- If it is, the Flash Menu program shows its display, allowing you to select or re-flash a core.
- If NO SCROLL is not being held down, the Flash Menu program checks if Flash Slot 1 contains a valid core.
- If it does, then the Flash Menu program attempts to load that core.
- If it succeeds, then the system reconfigures itself for that core, after which the behaviour of the system is according to that core.
- If it fails, the keyboard will go into "ambulance mode", showing flashing blue lights to indicate that some first-aid is required. Note that in ambulance mode the reset button has no effect: You must switch the MEGA65 off and on again.

If you have selected a different core in the Flash Menu, the process is similar, except that the ambulance lights will appear for only a limited time, as the FPGA will automatically search through the flash memory until it finds a valid core. If it gets to the end of the flash memory, it will start the MEGA65 Factory Core from slot 0 again.

DIP SWITCHES

The MEGA65 motherboard has 4 DIP switches (at position S3 on the PCB), some of which impact flashing and booting behaviour, so they are documented below:

Switch#	Purpose
1	Off = Disable Keyboard Flash On = Enable Keyboard Flash
2	Off = Enable audio on digital video connection On = Disable audio on digital video connection
3	Off = Disallow non-hypervisor access to QSPI flash On = Permit non-hypervisor access to QSPI flash
4	Off = Boot from slot 1 On = Boot from slot 2

CHAPTER 7

Floppy Disks, the Freezer, and D81 Images

- Disk Drives
- The Freezer
- Using D81 images

DISK DRIVES

The MEGA65 is compatible with a wide range of floppy disk drives, and can be configured to use them in a variety of different ways. It includes an internal 3.5" drive, which is compatible with double density and high density 3.5" disks. But did you know you can also use 5.25" disks? By connecting a compatible IEC drive to the rear Disk Drive/IEC Connector, you can use your MEGA65 with two or more physical floppy drives simultaneously!

There are also IEC drive emulators available, such as the SD2IEC. These devices (and the MEGA65) use "disk images", which are files that represent a floppy disk, usually stored on an SD card. By "mounting" a disk image file to a disk drive emulator, the MEGA65 can then use these files without realising it's actually being emulated. As far as the MEGA65 is concerned, it's using a real drive.

This allows you to use modern media to read and write your programs, and in some cases can drastically reduce read and write times. These files usually have the extension **.d64** (named after the C64), which are files that represent a *single* side of a 5.25" disk, or **.d81**, which represents an entire 3.5" disk (named after the 1581 floppy disk drive).

The MEGA65 can use either real disks and drives, **.d81** files, or a combination of both.

DISK DRIVE TERMINOLOGY

Before you get to know how to use the various disk drives that work with the MEGA65, it's important to be aware of some specific terminology which CBM computers and the MEGA65 use. This includes **BASIC** commands (such as **LOAD** and **SAVE**), and **CBDOS** (Computer Based Disk Operating System), which use the following:

UNIT is a device number in the range of 0-31. The numbers from 0 to 11 are reserved for the following device types:

Unit #	Device	Notes
0	Keyboard	Input
1	Unused	Was used for Cassettes on the C64
2	Unused	Was used for RS-232 on the C64
3	Screen	Input/Output
4-5	IEC Printer	Output
6-7	IEC Plotter	Output
8-9	CBDOS drives[1]	Internal floppy drive, 1565[2], or disk image
10-11	IEC drives	1541, 1571, 1581, FD-2000

[1] IEC drives can be assigned to devices 8-9, by moving the built-in MEGA65 devices to 10-11 first. Once the drives have been reassigned, the IEC drives can be switched on.

[2] The 1565 was a prototype external C65 disk drive that never made it to production.

DRIVE is the drive number of a **UNIT**:

Unit	Drive Numbers	Comment
1541 IEC	0	Single drive
1571 IEC	0	Single drive
1581 IEC	0	Single drive
FD-2000 IEC	0	Single drive (CMD)
FD-4000 IEC	0	Single drive (CMD)
SD2IEC	0	Disk images
4040 IEEE-488	0,1	Dual drive (CMD)
8050 IEEE-488	0,1	Dual drive (CMD)
8250 IEEE-488	0,1	Dual drive (CMD)

For all single drives, the drive number is always 0. This includes all of the well known drives with an IEC interface.

Dual disk drives (which use drive numbers 0 and 1) are usually equipped with an IEEE-488 interface, and need an IEEE-488 to IEC converter to be used on the MEGA65.

The internal floppy controller of the MEGA65 can be used to control two floppy drives. However, both drives must be attached to the same ribbon cable.

BASIC commands that address files or disks include **U** for UNIT and **D** for drive, which use the default values of **UNIT** = **8**, and **DRIVE** = **0**.

THE FREEZER

The MEGA65 **FREEZER** is a tool for changing system parameters at any time, regardless of the currently running program. It's very similar to the Freezer that many popular cartridges for the C64 included, such as the Action Replay cartridge, and The Final cartridge. Freezers generally work by taking over the CPU whilst leaving RAM intact. This way you can perform functions and change configuration settings without affecting the currently running program. Once you have finished using the Freezer, CPU control is given back to the original program.

The Freezer is invoked by pressing and holding RESTORE for half to one second. The current status of the computer is frozen and the Freezer menu, similar to the following picture is displayed. This chapter focuses on how to assign disk images and configure the internal floppy disk drive, but the other options in the Freezer are self explanatory, or will be covered in detail in online documentation.

The bottom-right part of the Freezer screen shows the current disk drive assignments. The internal CBDOS which the MEGA65 uses supports up to two 3.5" disk drives and .d81 images, in any combination. Drive 0 can be assigned to the internal floppy disk drive or a .d81 disk image, whereas Drive 1 can be assigned to an external floppy disk drive (connected to the same ribbon cable as the internal drive), or a .d81 disk image.

A typical configuration is usually one of the following:

Drive 0 Assignment	Drive 1 Assignment
Internal floppy disk drive	Disk image
Disk image	Disk image

The assignment, and mounting of disk images can be performed by pressing **0** for drive 0, or **1** for drive 1.

You may have noticed that the drive numbers here (0-1) are different to the more commonly used **UNIT** numbers (8-9), which you may be more used to. The drive numbers of 0 and 1 are used internally by CBDOS. However, **BASIC** and Kernal address these storage devices by **UNIT** numbers. CBDOS emulates two single drives for the operating system, by assigning separate unit numbers to drive 0 and drive 1. The default assignment is:

Unit/Drive	Assignment
UNIT 8, DRIVE 0	Internal drive 0 (internal floppy or disk image)
UNIT 9, DRIVE 0	Internal drive 1 (external floppy or disk image)

You may wish to change this unit assignment, if for example a 1541 drive is connected to the IEC port as unit 8. In this case, the internal drive assignment can be switched to an alternative unit to avoid conflict. To do this, press **8** to tog-

gle the unit assignment of drive 0 between 8 and 10, and **9** to toggle the unit assignment of drive 1, between 9 and 11.

After setting the preferences, the Freezer can be exited with **F3**. The Freezer restores the screen and returns to the interrupted program.

Note: The drive and unit assignments are temporary, and will be reset to their defaults after power down or reset. For permanent settings, use the **CONFIGURE** menu. You can read more about this menu on page 35.

USING D81 IMAGES

As previously mentioned, the MEGA65 allows the use of D81 files instead of using physical disks. The MEGA65 includes a built-in file browser, which allows you to choose which D81 file to mount. To do this, enter the Freezer menu, then press **0** or **1**, to choose the drive to mount it to.

When you do this, the file browser will appear:

The file browser will list all D81 images on the microSD card. The MEGA65 prioritises the external microSD card, so if you prefer to use an image on the internal SD card, the microSD card should be removed (when doing this, the MEGA65 needs to be reset as well).

Press the cursor keys to choose a file. If the currently selected file isn't a valid D81, the border will be red, otherwise it will be blue, as shown above. Whilst browsing for D81 files, the list of files within the currently selected D81 file will be shown on the right-hand side.

You may have noticed that the file browser has a **- NO DISK -**, **- NEW D81 IMAGE -** and a **- INTERNAL 3.5" -** entry. **- NO DISK -** is used to tell

the MEGA65 that no disk is inserted, and to not try and find one when performing a disk-based operation.

– NEW D81 IMAGE – creates a new, blank D81 image to use. When you select this option, you'll be asked to enter the filename for the new image, which is limited to 8 characters in length.

– INTERNAL 3.5" – is used when using a physical 3.5" disk in the internal drive. If no disk is inserted and a disk-based operation is performed with the **– INTERNAL 3.5" –** option, the MEGA65 will try and find a disk, which could take a few seconds. Any previously mounted D81 files will be unmounted when using these options.

When you find the D81 file you wish to mount, press RETURN. This will take you back to the Freezer menu, and the selected D81 file will be shown under the selected drive. In the following screenshot, the **MEGA65.D81** file has been mounted to drive 0:

Next, press F3 to exit the Freezer menu and return to the currently running program. Alternatively, pressing F5 will exit the Freezer, and reset the MEGA65 as well.

Now that the file has been mounted, you can use the **DIR** command to list the files in the image, or any of the **LOAD** and **SAVE** commands, like you would on a physical disk.

AUTO BOOTING FROM A DISK

If you would like the MEGA65 to automatically perform any BASIC commands upon startup, you can create an **AUTOBOOT.C65** PRG file on your disk/image. In this file, you can write BASIC programs to do things such as changing the **BACKGROUND**, **BORDER**, or **FOREGROUND** colours, or even **LOAD** a program.

For example, to change the background and border colours to black, and load a program called "MYPROGRAM" automatically, create a BASIC listing like the following, then **SAVE** it:

```
10 BACKGROUND 0
20 BORDER 0
30 LOAD "MYPROGRAM.PRG"
SAVE "AUTOBOOT.C65"
```

SAVING D81 FILE SETTINGS

The D81 settings are temporary, and will be reset to their defaults after power down. You can change the default settings in the **CONFIGURE** menu. When opening the Configure menu, press the F1 and F2 keys to reach the CHIPSET section, then navigate to the F011 DISK CONTROLLER options. Select the USES SDCARD DISK IMAGE setting, then enter the name of the disk image you wish to have mounted by default in the DEFAULT DISK IMAGE field. A screenshot showing these settings with MEGA65.D81 as the default D81 disk image is shown:

You can read more about the Configure menu on page 31.

APPENDIX A

BASIC 65 Command Reference

- Commands, Functions and Operators
- BASIC Command Reference

COMMANDS, FUNCTIONS AND OPERATORS

This appendix describes each of the commands, functions, and other callable elements of BASIC 65, which is an enhanced version of BASIC 10. Some of these can take one or more arguments, which are pieces of input that you provide as part of the command or function call, to help describe what you want to achieve. Some also require that you use special words.

Below is an example of how commands, functions, and operators (all of which are also known as keywords) will be described in this appendix.

KEY number**,** string

Here, **KEY** is a keyword. Keywords are special words that BASIC understands. Keywords are always written in **BOLD CAPITALS**, so that you can easily recognise them.

The words not in bold must be replaced for the command, function or operator to work. In this example, we need to replace number with a numeric expression, and string with a string expression. We'll explain what expressions are a bit more in a few moments.

The comma, and some other symbols and punctuation marks, just represent themselves when they are in bold. In our example here, it means that there must be a comma between the number and the string.

You might also see symbols and punctuation marks that are not in bold. When they are not in bold they have a special meaning. You might see square brackets around something. For example: [**,** numeric expression]. This means that whatever appears between the square brackets is optional. That is, you can include it if you need to, but the command, function or operator will also work just fine without it. For example, the **CIRCLE** command has an optional numeric argument to indicate if the circle should be filled when being drawn.

This arrangement of keywords, expressions and symbols is what's called syntax. If you miss something out, or put the wrong thing in the wrong place, it is called a syntax error. The computer will tell you that you have a syntax error by displaying a `?SYNTAX ERROR` message.

There is nothing to worry about if you get an error from the MEGA65. Instead, it is just the MEGA65's way of telling you that something isn't quite right, so that you can more easily find and fix the problem. Error messages such as this won't hurt the computer or cause any damage to your program, so there is nothing to worry about. For example, if we accidentally left the comma out, or replaced it with a full stop, the MEGA65 will respond with a `?SYNTAX ERROR`, similar to what's shown below:

```
KEY 8"FISH"

?SYNTAX ERROR

KEY 8,"FISH"

?SYNTAX ERROR
```

It is very common for commands, functions and operators to use one or more **"expressions"**. An expression is just a fancy name for something that has, or equates to a value. This could be as simple as a string ("HELLO"), a number (23.7), or a complex calculation that might include one or more functions or operators (LEN("HELLO") * (3 XOR 7)). Generally speaking, expressions can result in either a string or a numeric result. In this case we call the expressions string expressions or numeric expressions. For example, "HELLO" is a **string expression**, while 23.7 is a **numeric expression**.

It is important to use the correct type of expression when writing your programs. If you accidentally use the wrong type, the MEGA65 will display a ?TYPE MISMATCH ERROR, to say that the type of expression you gave doesn't match what it expected. For example, we will get a ?TYPE MISMATCH ERROR if we type the following command, because "POTATO" is a string expression instead of a numeric expression:

```
KEY "POTATO","SOUP"
```

If you wish, you can try typing this in yourself.

Commands are statements that you can use directly from the READY. prompt, or from within a program, for example:

```
PRINT "HELLO"
HELLO

10 PRINT "HELLO"
RUN
HELLO
```

You can place a sequence of statements within a single line by separating them with colons, for example:

```
PRINT "HELLO" : PRINT "HOW ARE YOU?" : PRINT "HOW IS THE WEATHER?"
HELLO
HOW ARE YOU?
HOW IS THE WEATHER?
```

DIRECT MODE COMMANDS

Note that some commands are said to only work in **direct mode**. This means that the command can't be part of a BASIC program, but can be entered directly to the screen. In the two **PRINT** examples above, the first was entered in direct mode, whereas the second one wasn't. The examples above would work since **PRINT** works in both direct and indirect mode.

COMMAND FORMAT SYNTAX

The following table describes what the other symbols found in this appendix mean.

Symbol	Meaning
...	The bracket can be repeated zero or more times
[ ]	Optional
< | >	Include one of the choices
[|]	Optionally include one of the choices
{ , }	One or more of the arguments is required. The commas to the left of the last argument included are required. Trailing commas must be omitted. See **CURSOR** for an example.
[{ , }]	Similar to { , } but all arguments can be omitted

FONTS

Whenever there's a piece of text in this appendix that reflects some logic, something you can type, or something the MEGA65 could display, the text will WILL USE THIS FONT. This helps make it easier to for you to distinguish between these things and the written text.

BASIC 65 CONSTANTS

Type	Example	Example
Decimal Integer	32000	-55
Decimal Fixed Point	3.14	-7654.321
Decimal Floating Point	1.5E03	7.7E-02
Hex	$D020	$FF
String	"X"	"TEXT"

BASIC 65 VARIABLES

Each scalar variable consumes 8 bytes of storage in memory. The reserved area in bank 0 from $F700-$FEFF can store 256 variables. Variables don't need to be declared, and their the type is determined by an appended character. All variables without an appended character are regarded as REAL by default, and storage is claimed at their first usage. They are also initialised to zero, whereas string variables are initialised as an empty string "".

Type	Appended Character	Range	Example
Byte	&	0 .. 255	BY& = 23
Integer	%	-32768 .. 32767	I% = 5
Real	none	-1E37 .. 1E37	XY = 1/3
String	$	length = 0 .. 255	AB$ = "TEXT"

BASIC 65 ARRAYS

Each array consumes the number of elements multiplied by the item size, plus the size of the header (6 + 2 * dimensions) in memory. For example the array

```
100 DIM X(8,2,3)
```

has 3 dimensions and 108 (9 x 3 x 4) items. You might be asking: Why is it 9 x 3 x 4, when the program uses 8, 2, and 3? This is because array indexes start at 0, not 1.

The size for real items is 5, so the data of that array above would occupy 540 (5 x 108) bytes. The header size is 12 bytes (6 + 2 * 3), so the total length in memory is 552 bytes (540 + 12).

Arrays are stored in bank 1 starting at address $2000 and expand upwards. They share the available memory at $2000 .. $F6FF with the string area, which starts in bank 1 at address $F6FF, and expands *downwards*. Each of the above scalar variable types can be used as an array, by declaring them with a **DIM** statement. The arrays are initialised to zero for all elements on declaration. If an undeclared array element is used, an automatic implicit declaration is performed, which sets the upper boundary for each dimension to 10. For example, the usage of an undeclared element `AB(3,5)` would automatically perform a `DIM AB(10,10)`. As noted previously, the lower boundary for each dimension is always 0 (zero), so an array initialised with `DIM AB(10)` consists of 11 elements and accepts indexes from 0 to 10.

String arrays are more precisely expressed as *arrays of string descriptors*. Each item consists of three bytes, which hold these values: The length of the string, and the 2 byte address (low/high byte) of the string in string memory. The usage of the BASIC function **POINTER** with a string or string array element as the argument, returns the address of the descriptor, not the string itself.

Type & Item Size		Appended Character	Range	Example
Byte Array	1	&	0 .. 255	BY&(5,6) = 23
Integer Array	2	%	-32768 .. 32767	I%(0,10) = 5
Real Array	5	none	-1E37 .. 1E37	XY(I%) = 1/3
String Array	3	$	0 .. 255 characters	AB$(X) = "TEXT"

BASIC 65 OPERATORS

BASIC 65 provides a set of operators that are typical of most BASIC programming dialects. The usage and precedence of these operators is documented in this section.

The = symbol is used both as an assignment operator, and as a relational operator for testing equality. For example, in the statement A = B = 5, the first equal sign is the assignment operator, while the second is a logical operator, comparing the variable B with 5. The value of A will either be assigned the value -1 (for **TRUE**), or the value 0 (for **FALSE**). You may have noticed that the value of -1 for **TRUE** is different to other programming languages, such as **C**, where the value of 1 is used for **TRUE** instead.

The + symbol can be used as a positive sign for numerical expressions, as an addition operator, or for string concatenation. The number and type of operands determines the operation.

The - symbol can be used as a negative sign for numerical expressions, or as a subtraction operator. The number and type of operands determines the operation.

The operators NOT, AND, OR and XOR can be used both as logical operators, or as boolean operators.

- Logical Operator Example: IF A>B AND A<0
- Boolean Operator Example: A = B AND $7F

Both examples always produce an integer result internally, which can be interpreted either numerically or logically. If the result of a comparison is **TRUE**, the value will be set to **-1**, while a **FALSE** result yields **0**. In the boolean operator example above, the AND operator converts both operands to a 16-bit integer value, and performs a bitwise **AND** for all 16 bits. This example will take the value of B, set the upper 9 bits to zero, and store the result in A.

The result of logical operations can be used in numerical expressions as well, for example, A = A - (B > 7) will increment A by 1 if the result of (B > 7) is **TRUE** (-1). This is because the mathematical expression of A = A - (-1) is the same as A = A + 1.

The operators have precedences, which are listed in the tables below. In the statement A * A - B * B both multiplications will be performed first, before the subtraction is executed. Parentheses are used to change the precedence, for example A * (A - B) * B) will execute the subtraction first.

ASSIGNMENT OPERATOR

Symbol	Description	Examples
=	Assignment	A = 42, A$ ="HELLO", A = B (42

UNARY MATHEMATICAL OPERATORS

Name	Symbol	Description	Example
Plus	+	Positive sign	A = +42
Minus	-	Negative sign	B = -42

BINARY MATHEMATICAL OPERATORS

Name	Symbol	Description	Example
Plus	+	Addition	A = B + 42
Minus	-	Subtraction	B = A - 42
Asterisk	*	Multiplication	C = A * B
Slash	/	Division	D = B / 13
Up Arrow	↑	Exponentiation	E = 2 ↑ 10

Note that the ↑ character used for exponentiation is entered with [↑], which is next to [RESTORE].

RELATIONAL OPERATORS

Symbol	Description	Example
)	Greater Than	A) 42
)=	Greater Than or Equal To	B)= 42
(	Less Than	A (42
(=	Less Than or Equal To	B (= 42
=	Equal	A = 42
()	Not Equal	B () 42

LOGICAL OPERATORS

Keyword	Description	Example
AND	And	A) 42 AND A (84
OR	Or	A) 42 OR A = 0
XOR	Exclusive Or	A) 42 XOR B) 42
NOT	Negation	C = NOT A) B

BOOLEAN OPERATORS

Keyword	Description	Example
AND	And	A = B AND $FF
OR	Or	A = B OR $80
XOR	Exclusive Or	A = B XOR 1
NOT	Negation	A = NOT 22

STRING OPERATOR

Name	Symbol	Description	Operand type	Example
Plus	+	Concatenates Strings	String	A$ = B$ + ".PRG"

OPERATOR PRECEDENCE

Precedence	Operators
High	↑
	+ - (Unary Mathematical)
	* /
	+ - (Binary Mathematical)
	< <= > >= = <>
	NOT
	AND
Low	OR XOR

BASIC COMMAND REFERENCE

ABS

Format: **ABS**(x)

Usage: **ABS** returns the absolute value of the numeric argument **x**.

x numeric argument (integer or real expression).

Remarks: The result is of type real.

Example: Using **ABS**

```
PRINT ABS(-123)
123
PRINT ABS(4.5)
4.5
PRINT ABS(-4.5)
4.5
```

AND

Format: operand **AND** operand

Usage: **AND** performs a bit-wise logical AND operation on two 16-bit values. Integer operands are used as they are. Real operands are converted to a signed 16-bit integer (losing precision). Logical operands are converted to 16-bit integer using $FFFF (decimal -1) for TRUE, and $0000 (decimal 0) for FALSE.

Expression	Result
0 AND 0	0
0 AND 1	0
1 AND 0	0
1 AND 1	1

Remarks: The result is of type integer. If the result is used in a logical context, the value of 0 is regarded as FALSE, and all other non-zero values are regarded as TRUE.

Examples: Using **AND**

```
PRINT 1 AND 3
1
PRINT 128 AND 64
0
```

In most cases, **AND** is used in **IF** statements.

```
IF (C )= 0 AND C ( 256) THEN PRINT "BYTE VALUE"
```

APPEND

Format: **APPEND#** channel**,** filename [**,D** drive] [**,U** unit]

Usage: Opens an existing sequential file of type **SEQ** or **USR** for writing, and positions the write pointer at the end of the file.

channel number, where:

- **1 <= channel <= 127** line terminator is CR.
- **128 <= channel <= 255** line terminator is CR LF.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **APPEND#** works similarly to **DOPEN#... ,W**, except that the file must already exist. The content of the file is retained, and all printed text is appended to the end. Trying to **APPEND** to a non existing file reports a DOS error.

Examples: Open existing file in append mode:

```
APPEND#5,"DATA",U9
APPEND#130,(DD$),U(UN%)
APPEND#3,"USER FILE,U"
APPEND#2,"DATA BASE"
```

ASC

Format: **ASC**(string)

Usage: Takes the first character of the string argument and returns its numeric code value. The name was apparently chosen to be a mnemonic to ASCII, but the returned value is in fact the so-called PETSCII code.

Remarks: **ASC** returns zero for an empty string, whose behaviour is different to BASIC 2, where ASC("") gave an error. The inverse function to **ASC** is **CHR$**. Refer to the **CHR$** function on page 92 for more information.

Examples: Using **ASC**

```
PRINT ASC("MEGA")
77
PRINT ASC("")
0
```

ATN

Format: **ATN**(numeric expression)

Usage: Returns the arc tangent of the argument. The result is in the range $(-\pi/2$ to $\pi/2)$

Remarks: A multiplication of the result with $180/\pi$ converts the value to the unit "degrees". **ATN** is the inverse function to **TAN**.

Examples: Using **ATN**

```
PRINT ATN(0.5)
 .463647609
PRINT ATN(0.5) * 180 / π
 26.5650512
```

AUTO

Format: **AUTO** [step]

Usage: Enables faster typing of BASIC programs. After submitting a new program line to the BASIC editor with RETURN, the **AUTO** function generates a new BASIC line number for the entry of the next line. The new number is computed by adding **step** to the current line number.

step line number increment

Typing **AUTO** with no argument disables it.

Examples: Using **AUTO**

```
AUTO 10  : USE AUTO WITH INCREMENT 10
AUTO     : SWITCH AUTO OFF
```

BACKGROUND

Format: **BACKGROUND** colour

Usage: Sets the background colour of the screen to the argument, which must be in the range of 0 to 255. All colours within this range are customisable via the **PALETTE** command. On startup, the MEGA65 only has the first 32 colours configured, which are described in the following table.

Colours: **Index and RGB values of colour palette**

Index	Red	Green	Blue	Colour
0	0	0	0	Black
1	15	15	15	White
2	15	0	0	Red
3	0	15	15	Cyan
4	15	0	15	Purple
5	0	15	0	Green
6	0	0	15	Blue
7	15	15	0	Yellow
8	15	6	0	Orange
9	10	4	0	Brown
10	15	7	7	Pink
11	5	5	5	Dark Grey
12	8	8	8	Medium Grey
13	9	15	9	Light Green
14	9	9	15	Light Blue
15	11	11	11	Light Grey
16	14	0	0	Guru Meditation
17	15	5	0	Rambutan
18	15	11	0	Carrot
19	14	14	0	Lemon Tart
20	7	15	0	Pandan
21	6	14	6	Seasick Green
22	0	14	3	Soylent Green
23	0	15	9	Slimer Green
24	0	13	13	The Other Cyan
25	0	9	15	Sea Sky
26	0	3	15	Smurf Blue
27	0	0	14	Screen of Death
28	7	0	15	Plum Sauce
29	12	0	15	Sour Grape
30	15	0	11	Bubblegum
31	15	3	6	Hot Tamales

Example: Using **BACKGROUND**

```
BACKGROUND  3 : REM SELECT BACKGROUND COLOUR CYAN
```

BACKUP

Format: **BACKUP U** source **TO U** target
BACKUP D source **TO D** target [**,U** unit]

Usage: The first form of **BACKUP**, specifying units for source and target can only be used for the drives connected to the internal FDC (Floppy

Disk Controller). Units 8 and 9 are reserved for this controller. These can be either the internal floppy drive (unit 8) and another floppy drive (unit 9) attached to the same ribbon cable, or mounted D81 disk images. Therefore, **BACKUP** can be used to copy from floppy to floppy, floppy to image, image to floppy and image to image, depending on image mounts and the existence of a second physical floppy drive.

The second form of **BACKUP**, specifying drives for source and target, is meant to be used for dual drive units connected to the IEC bus. For example: CBM 4040, 8050, 8250 via an IEEE-488 to IEC adapter. The backup is then done by the disk unit internally.

source unit or drive # of source disk.
target unit or drive # of target disk.

Remarks: The target disk will be formatted and an identical copy of the source disk will be written.
BACKUP cannot be used to backup from internal devices to IEC devices or vice versa.

Examples: Using **BACKUP**

```
BACKUP U8 TO U9      : REM BACKUP INTERNAL DRIVE 8 TO DRIVE 9
BACKUP U9 TO U8      : REM BACKUP DRIVE 9 TO INTERNAL DRIVE 8
BACKUP D0 TO D1, U10 : REM BACKUP ON DUAL DRIVE CONNECTED VIA IEC
```

BANK

Format: **BANK** bank number

Usage: Selects the memory configuration for BASIC commands that use 16-bit addresses. These are **LOAD**, **LOADIFF**, **PEEK**, **POKE**, **SAVE**, **SYS**, and **WAIT**. Refer to the system memory map in the **MEGA65 Book** for more information.

Remarks: A value > 127 selects memory mapped I/O. The default value for the bank number is 128. This configuration has RAM from $0000 to $1FFF, the BASIC and KERNAL ROM, and I/O from $2000 to $FFFF.

Example: Using **BANK**

```
BANK 1   :REM SELECT MEMORY CONFIGURATION 1
```

BEGIN

Format: **BEGIN** ... **BEND**

Usage: **BEGIN** and **BEND** act as a pair of braces around a compound statement to be executed after **THEN** or **ELSE**. This overcomes the single line limitation of the standard **IF** ... **THEN** ... **ELSE** clause.

Remarks: Do not jump with **GOTO** or **GOSUB** into a compound statement, as it may lead to unexpected results.

Example: Using **BEGIN** and **BEND**

```
10 GET A$
20 IF A$>="A" AND A$<="Z" THEN BEGIN
30 PW$=PW$+A$
40 IF LEN(PW$)>7 THEN 90
50 BEND :REM IGNORE ALL EXCEPT (A-Z)
60 IF A$<>CHR$(13) GOTO 10
90 PRINT "PW=";PW$
```

BEND

Format: **BEGIN** ... **BEND**

Usage: **BEGIN** and **BEND** act as a pair of braces around a compound statement to be executed after **THEN** or **ELSE**. This overcomes the single line limitation of the standard **IF** ... **THEN** ... **ELSE** clause.

Remarks: The example below shows a quirk in the implementation of the compound statement. If the condition evaluates to **FALSE**, execution does not resume right after **BEND** as it should, but at the beginning of the next line. Test this behaviour with the following program:

Example: Using **BEGIN** and **BEND**

```
10 IF Z > 1 THEN BEGIN:A$="ONE"
20 B$="TWO"
30 PRINT A$;" ";B$;:BEND:PRINT " QUIRK"
40 REM EXECUTION RESUMES HERE FOR Z <= 1
```

BLOAD

Format: **BLOAD** filename [**,B** bank] [**,P** address] [**,R**] [**,D** drive] [**,U** unit]

Usage: "Binary LOAD" loads a file of type **PRG** into RAM at address P.

BLOAD has two modes: The flat memory address mode can be used to load a program to any address in the 28-bit (256MB) address range where RAM is installed. This includes the standard RAM banks 0 to 5, as well as the 8MB of "attic RAM" at address $8000000.

This mode is triggered by specifying an address at parameter P that is larger than $FFFF. The bank parameter is ignored in this mode.

For compatibility reasons with older BASIC versions, **BLOAD** accepts the syntax with a 16-bit address at P and a bank number at B as well. The attic RAM is out of range for this compatibility mode.

The optional parameter **R** (RAW MODE) does not interpret or use the first two bytes of the program file as the load address, which is otherwise the default behaviour. In RAW MODE every byte is read as data.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

bank specifies the RAM bank to be used. If not specified, the current bank, as set with the last **BANK** statement will be used.

address can be used to override the load address that is stored in the first two bytes of the **PRG** file.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **BLOAD** cannot cross bank boundaries.

BLOAD uses the load address from the file, if no P parameter is given.

Examples: Using **BLOAD**

```
BLOAD "ML DATA", B0, U9
BLOAD "SPRITES"
BLOAD "ML ROUTINES", B1, P32768
BLOAD (FI$), B(BA%), P(PA), U(UN%)
BLOAD "CHUNK",P($8000000)          :REM LOAD TO ATTIC RAM
```

BOOT

Format: **BOOT** filename [**,B** bank] [**,P** address] [**,D** drive] [**,U** unit]
BOOT SYS
BOOT

Usage: **BOOT filename** loads a file of type **PRG** into RAM at address P and bank B, and starts executing the code at the load address.

BOOT SYS loads the boot sector from sector 0, track 1 and unit 8 to address $0400 in bank 0, and performs a `JSR $0400` afterwards (Jump To Subroutine).

BOOT with no parameters attempts to load and execute a file named AUTOBOOT.C65 from the default unit 8. It's short for **RUN "AUTOBOOT.C65"**.

filename is either a quoted string such as `"DATA"`, or a string expression in brackets such as `(FI$)`.

bank specifies the RAM bank to be used. If not specified, the current bank, as set with the last **BANK** statement, will be used.

address can be used to override the load address, that is stored in the first two bytes of the **PRG** file.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **BOOT SYS** copies the contents of one physical sector (two logical sectors) = 512 bytes from disk to RAM, filling RAM from $0400 to $05FF.

Examples: Using **BOOT**

```
BOOT $Y$
BOOT (FI$), B(BA%), P(PA), U(UN%)
BOOT
```

BORDER

Format: **BORDER** colour

Usage: Sets the border colour of the screen to the argument, which must be in the range of 0 to 255. All colours within this range are cus-

tomisable via the **PALETTE** command. On startup, the MEGA65 only has the first 32 colours configured, which are described in the table under **BACKGROUND** on page 79.

Example: Using **BORDER**

```
10 BORDER  4 : REM SELECT BORDER COLOUR PURPLE
```

BOX

Format: **BOX** x0,y0, x2,y2 [, solid]
BOX x0,y0, x1,y1, x2,y2, x3,y3 [, solid]

Usage: The first form of **BOX** with two coordinate pairs and an optional **solid** parameter draws a simple rectangle, assuming that the coordinate pairs declare two diagonally opposite corners.

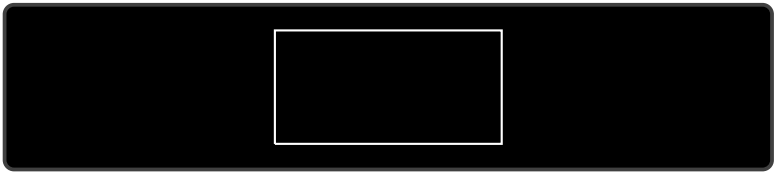
The second form with four coordinate pairs declares a path of four points, which will be connected with lines. The path is closed by connecting the last coordinate with the first.

The quadrangle is drawn using the current drawing context set with **SCREEN**, **PALETTE** and **PEN**. The quadrangle is filled if the parameter **solid** is not 0.

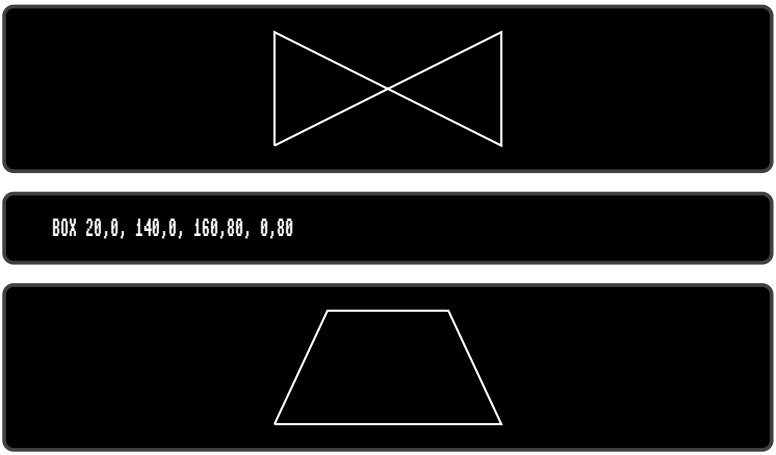
Remarks: **BOX** can be used with four coordinate pairs to draw any shape that can be defined with four points, not only rectangles. For example rhomboids, kites, trapezoids and parallelograms. It is also possible to draw bow tie shapes.

Examples: Using **BOX**

```
BOX 0,0, 160,0, 160,80, 0,80
```



```
BOX 0,0, 160,80, 160,0, 0,80
```



BSAVE

Format: **BSAVE** filename**, P** start **TO P** end [**,B** bank] [**,D** drive] [**,U** unit]

Usage: "Binary SAVE" saves a memory range to a file of type **PRG**.

BSAVE has two modes: The flat memory address mode can be used to save a memory block in the 28-bit (256MB) address range where RAM is installed. This includes the standard RAM banks 0 to 5, as well as the 8MB of "attic RAM" at address $8000000.

This mode is triggered by specifying addresses for the start and end parameter P, that are larger than $FFFF. The bank parameter is ignored in this mode. This flat memory mode allows saving ranges greater than 64K.

For compatibility reasons with older BASIC versions, **BSAVE** accepts the syntax with 16-bit addresses at P and a bank number at B as well. The attic RAM is out of range for this compatibility mode. This mode cannot cross bank boundaries, so start and end address must be in the same bank.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI$). If the first character of the filename is an at sign '@', it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS.

start is the first address, where the saving begins. It also becomes the load address, which is stored in the first two bytes of the **PRG** file.

end address where the saving ends. **end-1** is the last address to be used for saving.

bank specifies the RAM bank to be used. If not specified, the current bank, as set with the last **BANK** statement, will be used.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: The length of the file is **end - start + 2**.
If the number after an argument letter is not a decimal number, it must be set in parenthesis, as shown in the third and fourth line of the examples.

The **PRG** file format that is used by **BSAVE** requires the load address to be written to the first two bytes. If the saving is done with a bank number that is not zero, or a start address greater than $FFFF, this information will not fit. For compatibility reasons, only the the two low order bytes are written. Loading the file with the **BLOAD** command will then require the full 16-bit range of the load address as a parameter.

Examples: Using **BSAVE**

```
BSAVE "ML DATA", P 32768 TO P 33792, B0, U9
BSAVE "SPRITES", P 1536 TO P 2058
BSAVE "ML ROUTINES", B1, P($9000) TO P($A000)
BSAVE (FI$), B(BA%), P(PA) TO P(PE), U(UN%)
```

BUMP

Format: **BUMP**(type)

Usage: Used to detect sprite-sprite (type=1) or sprite-data (type=2) collisions. The return value is an 8-bit mask with one bit per sprite. The bit position corresponds to the sprite number. Each bit set in the returned value indicates that the sprite for its position was involved in a collision since the last call of **BUMP**. Calling **BUMP** resets the collision mask, so you will always get a summary of collisions encountered since the last call of **BUMP**.

Remarks: It's possible to detect multiple collisions, but you will need to evaluate the sprite coordinates to detect which sprites have collided.

Example: Using **BUMP**

```
10 S% = BUMP(1) : REM SPRITE-SPRITE COLLISION
20 IF (S% AND 6) = 6 THEN PRINT "SPRITE 1 & 2 COLLISION"
30 REM ---
40 S% = BUMP(2) : REM SPRITE-DATA COLLISION
50 IF (S% <> 0) THEN PRINT "SOME SPRITE HIT DATA REGION"
```

Sprite	Return	Mask
0	1	0000 0001
1	2	0000 0010
2	4	0000 0100
3	8	0000 1000
4	16	0001 0000
5	32	0010 0000
6	64	0100 0000
7	128	1000 0000

BVERIFY

Format: **BVERIFY** filename [**,P** address] [**,B** bank] [**,D** drive] [**,U** unit]

Usage: "Binary VERIFY" compares a memory range to a file of type **PRG**.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (F1$).

bank specifies the RAM bank to be used. If not specified, the current bank, as set with the last **BANK** statement, will be used.

address is the address where the comparison begins. If the parameter P is omitted, it is the load address that is stored in the first two bytes of the **PRG** file that will be used.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **BVERIFY** can only test for equality. It gives no information about the number, or position of different valued bytes. In direct mode **BVERIFY** exits either with the message OK or with VERIFY ERROR. In program mode, a VERIFY ERROR either stops execution or enters the **TRAP** error handler, if active.

Examples: Using **BVERIFY**

```
BVERIFY "ML DATA", P 32768, B0, U9
BVERIFY "SPRITES", P 1536
BVERIFY "ML ROUTINES", B1, P(DEC("9000"))
BVERIFY (FI$), B(BA%), P(PA), U(UN%)
```

CATALOG

Format: **CATALOG** [filepattern] [**,W**] [**,R**] [**,D** drive] [**,U** unit]
$ [filepattern] [**,W**] [**,R**] [**,D** drive] [**,U** unit]

Usage: Prints a file catalog/directory of the specified disk.

The **W** (Wide) parameter lists the directory three columns wide on the screen and pauses after the screen has been filled with a page (63 directory entries). Pressing any key displays the next page.

The **R** (Recoverable) parameter includes files in the directory which are flagged as deleted but still recoverable.

filepattern is either a quoted string, for example: `"DA*"` or a string expression in brackets, e.g. `(DI$)`

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **CATALOG** is a synonym of **DIRECTORY** and **DIR**, and produces the same listing. The **filepattern** can be used to filter the listing. The wildcard characters * and **?** may be used. Adding **,T=** to the pattern string, with **T** specifying a filetype of **P**, **S**, **U** or **R** (for **P**RG, **S**EQ, **U**SR, **R**EL) filters the output to that filetype.

The shortcut symbol **$** can only be used in direct mode.

Examples: Using **CATALOG**

```
CATALOG
0 "BLACK SMURF     " BS 2A
508  "STORY PHOBOS"    SEQ
27   "C8096"           PRG
25   "C128"            PRG
104 BLOCKS FREE.
```

```
CATALOG "*,T=S"
0 "BLACK SMURF     " BS 2A
508  "STORY PHOBOS"    SEQ
104 BLOCKS FREE.
```

Below is an example showing how a directory looks with the **w**ide parameter:

```
DIR W
0 "BASIC EXAMPLES  "
1 "BEGIN"          P    1 "FREAD"          P    2 "PAINT.COR"      P
1 "BEND"           P    1 "FRE"            P    3 "PALETTE.COR"    P
1 "BUMP"           P    2 "GET#"           P    1 "PEEK"           P
1 "CHAR"           P    1 "GETKEY"         P    3 "PEN"            P
1 "CHR$"           P    1 "GET"            P    1 "PLAY"           P
4 "CIRCLE"         P    2 "GOSUB"          P    2 "POINTER"        P
1 "CLOSE"          P    2 "GOTO.COR"       P    1 "POKE"           P
1 "CLR"            P    2 "GRAPHIC"        P    1 "POS"            P
2 "COLLISION"      P    1 "HELP"           P    1 "POT"            P
1 "CURSOR"         P    1 "IF"             P    1 "PRINT#"         P
0 "DATA BASE"      R    2 "INPUT#"         P    1 "PRINT"          P
1 "DATA"           P    2 "INPUT"          P    1 "RCOLOR.COR"     P
1 "DEF FN"         P    2 "JOY"            P    1 "READ"           P
1 "DIM"            P    1 "LINE INPUT#"    P    1 "RECORD"         P
1 "DO"             P    3 "LINE"           P    1 "REM"            P
5 "ELLIPSE"        P    1 "LOOP"           P    1 "RESTORE"        P
1 "ELSE"           P    1 "MID$"           P    1 "RESUME"         P
1 "EL"             P    1 "MOD"            P    1 "RETURN"         P
1 "ENVELOPE"       P    1 "MOVSPR"         P    1 "REVERS"         S
2 "EXIT"           P    1 "NEXT"           P    3 "RGRAPHIC"       P
1 "FOR"            P    2 "ON"             P    1 "RMOUSE"         P
```

CHANGE

Format: **CHANGE /**findstring**/ TO /**replacestring**/** [, line range]
CHANGE "findstring**" TO "**replacestring**"** [, line range]

Usage: **CHANGE** performs a **find and replace** of the BASIC program that is currently in memory. An optional **line range** limits the search to this range, otherwise the entire BASIC program is searched. At each occurrence of the **findstring**, the line is listed and the user is prompted for an action:

- **Y** **RETURN** perform the replace and find the next string

- N RETURN do **not** perform the replace and find the next string
- * RETURN replace the current and all following matches
- RETURN exit the command, and don't replace the current match

Remarks: Any un-shifted character that is not part of the string can be used instead of **/**.

However, using the double quote character finds text strings that are not tokenised, and therefore not part of a keyword.
For example, `CHANGE "LOOP" TO "OOPS"` will not find the BASIC keyword `LOOP`, because the keyword is stored as a token and not as text. However `CHANGE /LOOP/ TO /OOPS/` will find and replace it (possibly causing `SYNTAX ERROR`s).

Can only be used in direct mode.

Examples: Using **CHANGE**

```
CHANGE "XX$" TO "UU$", 2000-2700
CHANGE /IN/ TO /OUT/
CHANGE &IN& TO &OUT&
```

CHAR

Format: **CHAR** column**,** row**,** height**,** width**,** direction**,** string [**,** address of character set]

Usage: Displays text on a graphic screen. It can be used in all resolutions.

column (in units of character positions) is the start position of the output horizontally. As each column unit is 8 pixels wide, a screen width of 320 has a column range of 0-39, while a screen width of 640 has a column range of 0-79.

row (in pixel units) is the start position of the output vertically. In contrast to the column parameter, its unit is in pixels (not character positions), with the top row having the value of 0.

height is a factor applied to the vertical size of the characters, where 1 is normal size (8 pixels), 2 is double size (16 pixels), and so on.

width is a factor applied to the horizontal size of the characters, where 1 is normal size (8 pixels) 2 is double size (16 pixels), and so on. **direction** controls the printing direction:

- **1** up

- **2** right
- **4** down
- **8** left

The optional **address of character set** can be used to select a character set, different to the default character set at $29800, which includes upper and lower case characters.

Three character sets (see also **FONT**) are available:

- **$29000** Font A (ASCII)
- **$3D000** Font B (Bold)
- **$2D000** Font C (CBM)

The first part of the font (upper case / graphics) is stored at $xx000 - $xx7FF.

The second part of the font (lower case / upper case) is stored at $xx800 - $xxFFF.

string is a string constant or expression which will be printed. This string may optionally contain one or more of the following control characters:

Expression	Keyboard Shortcut	Description
CHR$(6)	CTRL+F	Flip Character
CHR$(18)	RVSON	Reverse
CHR$(146)	RVSOFF	Reverse Off
CHR$(21)	CTRL+U	Underline
CHR$(25)+"-"	CTRL+Y + "-"	Rotate Left
CHR$(25)+"+"	CTRL+Y + "+"	Rotate Right
CHR$(26)	CTRL+Z	Mirror

Remarks: Regular text mode control characters, such as cursor movement codes, will be ignored (neither printed nor interpreted).

Notice that the start position of the string has different units in the horizontal and vertical directions. Horizontal is in columns and vertical is in pixels.

Refer to the **CHR$** function on page 92 for more information.

Example: Using **CHAR**

```
10 SCREEN 640,400,2
20 CHAR 28,180,4,4,2,"MEGA65",$29000
30 GETKEY A$
40 SCREEN CLOSE
```

Will print the text "MEGA65" at the centre of a 640 x 400 graphic screen.

CHR$

Format: **CHR$(**numeric expression**)**

Usage: Returns a string containing one character, whose PETSCII value is equal to the argument.

Remarks: The argument range is from 0 - 255, so this function may also be used to insert control codes into strings. Even the NULL character, with code 0, is allowed.
CHR$ is the inverse function to **ASC**. The complete table of characters (and their PETSCII codes) is on page 217.

Example: Using **CHR$**

```
10 QUOTE$   = CHR$(34)
20 ESCAPE$  = CHR$(27)
30 PRINT QUOTE$;"MEGA65";QUOTE$ : REM PRINT "MEGA65"
40 PRINT ESCAPE$;"Q";       : REM CLEAR TO END OF LINE
```

CIRCLE

Format: **CIRCLE** xcentre**,** ycentre**,** radius [**,** solid]

Usage: A special case of **ELLIPSE**, using the same value for horizontal and vertical radius.

xcentre x coordinate of the centre in pixels

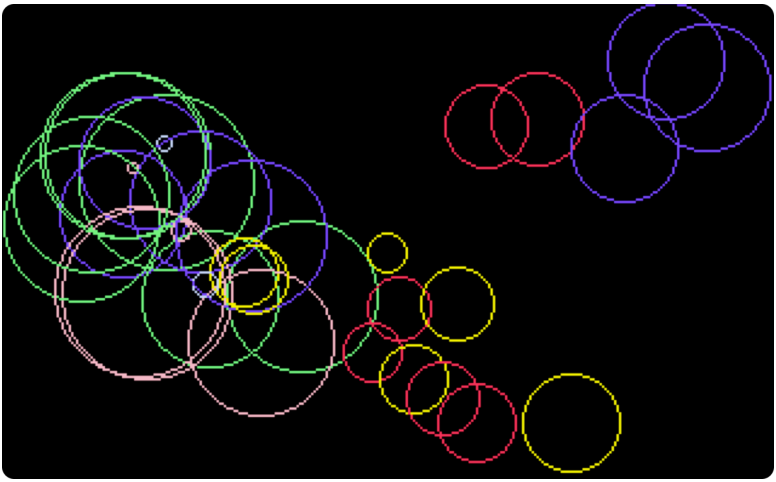
ycentre y coordinate of the centre in pixels

radius radius of the circle in pixels

solid fills the circle, if not zero

Remarks: **CIRCLE** is used to draw circles on screens with an aspect ratio of 1:1 (for example: 320 x 200 or 640 x 400). Whilst using other resolutions (such as 640 x 200), the shape will be an ellipse instead.

The example program uses the random number function **RND** for circle colour, size and position. So it shows a different picture for each run.



Example: Using **CIRCLE**

```
100 REM CIRCLE (AFTER F.BOWEN)
110 BORDER 0                          :REM BLACK
120 SCREEN 320,200,4                  :REM SIMPLE SCREEN SETUP
130 PALETTE 0,0,0,0,0                 :REM BLACK
140 PALETTE 0,1,RND(.)*16,RND(.)*16,15  :REM RANDOM COLOURS
150 PALETTE 0,2,RND(.)*16,15,RND(.)*16
160 PALETTE 0,3,15,RND(.)*16,RND(.)*16
170 PALETTE 0,4,RND(.)*16,RND(.)*16,15
180 PALETTE 0,5,RND(.)*16,15,RND(.)*16
190 PALETTE 0,6,15,RND(.)*16,RND(.)*16
200 SCNCLR 0                          :REM CLEAR
210 FOR I=0TO32                       :REM CIRCLE LOOP
220 PEN 0,RND(.)*6+1                  :REM RANDOM PEN
230 R=RND(.)*36+1                     :REM RADIUS
240 XC=R+RND(.)*320:IF(XC+R)>319THEN240:REM X CENTRE
250 YC=R+RND(.)*200:IF(YC+R)>199THEN250:REM Y CENTRE
260 XC=XC+WT*320:YC=YC+HT*200
270 CIRCLE XC,YC,R,.                  :REM DRAW
280 NEXT
290 GETKEY A$                         :REM WAIT FOR KEY
300 SCREEN CLOSE:BORDER 6
```

CLOSE

Format: **CLOSE** channel

Usage: Closes an input or output channel.

channel number, which was given to a previous call of commands such as **APPEND**, **DOPEN**, or **OPEN**.

Remarks: Closing files that have previously been opened before a program has completed is very important, especially for output files. **CLOSE** flushes output buffers and updates the directory information on disks. Failing to **CLOSE** can corrupt files and disks. BASIC does NOT automatically close channels nor files when a program stops.

Example: Using **CLOSE**

```
10 OPEN 2,8,2,"TEST,S,W"
20 PRINT#2,"TESTSTRING"
30 CLOSE 2 : REM OMITTING CLOSE GENERATES A SPLAT FILE
```

CLR

Format: **CLR**
CLR variable

Usage: Used for management of BASIC variables, arrays and strings. The run-time stack pointers, and the table of open channels is reset. After executing **CLR** all variables and arrays will be undeclared. **RUN** performs **CLR** automatically.

CLR variable clears (zeroes) the variable. **variable** can be a numeric variable or a string variable, but not an array.

Remarks: **CLR** should not be used inside loops or subroutines, as it destroys the return address. After **CLR**, all variables are unknown and will be initialised when they are next used.

Example: Using **CLR**

```
10 A=5: P$="MEGA65"
20 CLR
30 PRINT A;P$
RUN

 0
```

CMD

Format: **CMD** channel [**,** string]

Usage: Redirects the standard output from screen to a channel. This enables you to print listings and directories to other output channels. It is also possible to redirect this output to a disk file, or a modem.

channel number, which was given to a previous call of commands such as **APPEND**, **DOPEN**, or **OPEN**.

The optional **string** is sent to the channel before the redirection begins and can be used, for example, for printer or modem setup escape sequences.

Remarks: The **CMD** mode is stopped with **PRINT#**, or by closing the channel with **CLOSE**. It is recommended to use **PRINT#** before closing to make sure that the output buffer has been flushed.

Example: Using **CMD** to print a program listing:

```
OPEN 1,4   :REM OPEN CHANNEL #1 TO PRINTER AT UNIT 4
CMD 1
LIST
PRINT#1
CLOSE 1
```

COLLECT

Format: **COLLECT** [**,D** drive] [**,U** unit]

Usage: Rebuilds the **BAM** (Block Availability Map) of a disk, deleting splat files (files which have been opened, but not properly closed) and marking unused blocks as free.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: While this command is useful for cleaning a disk from splat files, it is dangerous for disks with boot blocks or random access files. These blocks are not associated with standard disk files and will therefore be marked as free and may be overwritten by further disk write operations.

Examples: Using **COLLECT**

```
COLLECT
COLLECT U9
COLLECT D0, U9
```

COLLISION

Format: **COLLISION** type [**,** line number]

Usage: Enables or disables a user-programmed interrupt handler. A call without the line number argument disables the handler, while a call with line number enables it. After the execution of **COLLISION** with line number, a sprite collision of the same type, (as specified in the **COLLISION** call) interrupts the BASIC program and performs a **GOSUB** to **line number**, which is expected to contain the user code for handling sprite collisions. This handler must give control back with **RETURN**.

type specifies the collision type for this interrupt handler:

Type	Description
1	Sprite - Sprite Collision
2	Sprite - Data - Collision
3	Light Pen

linenumber must point to a subroutine which has code for handling sprite collision and ends with **RETURN**.

Remarks: It is possible to enable the interrupt handler for all types, but only one can execute at any time. An interrupt handler cannot be interrupted by another interrupt handler. Functions such as **BUMP**, **LPEN** and **RSPPOS** may be used for evaluation of the sprites which are involved, and their positions.

Info: **COLLISION** wasn't completed in BASIC 10, and a working implementation will be available in a future BASIC 65 update.

Example: Using **COLLISION**

```
10 COLLISION 1,70 : REM ENABLE
20 SPRITE 1,1 : MOVSPR 1,120,  0 : MOVSPR 1,0#5
30 SPRITE 2,1 : MOVSPR 2,120,100 : MOVSPR 2,180#5
40 FOR I=1 TO 50000:NEXT
50 COLLISION 1 : REM DISABLE
60 END
70 REM SPRITE (-) SPRITE INTERRUPT HANDLER
80 PRINT "BUMP RETURNS";BUMP(1)
90 RETURN: REM RETURN FROM INTERRUPT
```

COLOR

Format: **COLOR** colour
COLOR <**ON** | **OFF**>

Usage: The first format works in the same way as the **FOREGROUND**, i.e: sets the foreground colour (text colour) of the screen to the colour argument, which must be in the range of 0 to 31. Refer to the table under **BACKGROUND** on page 79 for the colour values and their corresponding colours.

The second format enables or disables handling of character attributes on screen. If **COLOR** is **ON**, the screen routines take care of both character RAM and attribute RAM (or colour RAM). For example, if the screen is scrolling text, the attributes are also scrolled, so each character keeps its attribute. If **COLOR** is **OFF**, the attribute is fixed and only character movement is performed for screen characters. This speeds up screen handling, which could be useful when moving characters with different colours is not required.

Example: Using **COLOR**

```
10 COLOR ON  : COLOR 2 : PRINT "COLOR ON AND RED"
20 COLOR OFF : COLOR 2 : PRINT "COLOR OFF (CAN'T CHANGE COLOUR)"
30 COLOR ON  : COLOR 3 : PRINT "COLOR ON AND CYAN"

READY.

RUN
COLOR ON AND RED
COLOR OFF (CAN'T CHANGE COLOUR)
COLOR ON AND CYAN
```

CONCAT

Format: **CONCAT** appendfile [**,D** drive] **TO** targetfile [**,D** drive] [**,U** unit]

Usage: **CONCAT** (concatenation) appends the contents of **appendfile** to the **targetfile**. Afterwards, **targetfile** contains the contents of both files, while **appendfile** remains unchanged.

appendfile is either a quoted string, for example: `"DATA"` or a string expression in brackets, for example: `(FI$)`

targetfile is either a quoted string, for example: `"SAFE"` or a string expression in brackets, for example: `(F$$)`

If the disk unit has dual drives, it is possible to apply **CONCAT** to files which are stored on different disks. In this case, it is necessary to specify the drive# for both files. This is also necessary if both files are stored on drive#1.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **CONCAT** is executed in the DOS of the disk drive. Both files must exist and no pattern matching is allowed. Only files of type **SEQ** may be concatenated.

Examples: Using **CONCAT**

```
CONCAT "NEW DATA" TO "ARCHIVE" ,U9
CONCAT "ADDRESS",D0 TO "ADDRESS BOOK",D1
```

CONT

Format: **CONT**

Usage: Used to resume program execution after a break or stop caused by an **END** or **STOP** statement, or by pressing RUN STOP. This is a useful debugging tool. The BASIC program may be stopped and variables can be examined, and even changed. The **CONT** statement resumes execution.

Remarks: **CONT** cannot be used if a program has stopped because of an error. Also, any editing of a program inhibits continuation. Stopping and continuation can spoil the screen output, and can also interfere with input/output operations.

Example: Using **CONT**

```
10 I=I+1:GOTO 10
RUN

BREAK IN 10
READY.
PRINT I
 947
CONT
```

COPY

Format: **COPY** source [**,D** drive] [**,U** unit] **TO** [target] [**,D** drive] [**,U** unit]

Usage: Copies the contents of **source** to **target**. It is used to copy either single files or, by using wildcard characters, multiple files.

source is either a quoted string, e.g. "DATA" or a string expression in brackets, e.g. (FI$).

target is either a quoted string, e.g. "BACKUP" or a string expression in brackets, e.g. (FS$)

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

If none or one unit number is given, or the unit numbers before and after the TO token are equal, **COPY** is executed on the disk drive itself, and the source and target files will be on the same disk.

If the source unit (before TO) is different to the target unit (after TO), **COPY** is executed in MEGA65 BASIC by reading the source files into a RAM buffer and writing to the target unit. In this case, the target file name cannot be chosen, it will be the same as the source filename. The extended unit-to-unit copy mode allows the copying of single files, pattern matching files or all files of a disk. Any combination of units is allowed, internal floppy, D81 disk images, IEC floppy drives such as the 1541, 1571, 1581, or CMD floppy and hard drives.

Remarks: The file types **PRG**, **SEQ** and **USR** can be copied. If source and target are on the same disk, the target filename must be different from the source file name.

COPY cannot copy **DEL** files, which are commonly used as titles or separators in disk directories. These do not conform to Commodore DOS rules and cannot be accessed by standard **OPEN** routines.

REL files cannot be copied from unit to unit.

Examples: Using **COPY**

```
COPY U8 TO U9           :REM COPY ALL FILES
COPY "CODES" TO "BACKUP" :REM COPY SINGLE FILE
COPY "*.TXT",U8 TO U9    :REM PATTERN COPY
COPY "M*",U9 TO U11      :REM PATTERN COPY
```

COS

Format: **COS**(numeric expression)

Usage: Returns the cosine of the argument. The argument is expected in units of radians. The result is in the range (-1.0 to +1.0)

Remarks: An argument in units of **degrees** can be converted to **radians** by multiplying it with $\pi/180$.

Examples: Using **COS**

```
PRINT COS(0.7)
 0.76484219

X=60:PRINT COS(X * π / 180)
 0.5
```

CURSOR

Format: **CURSOR** <**ON** | **OFF**> [{, column, row, style}]
CURSOR {column, row, style}

Usage: Moves the text cursor to the specified position on the current text screen.

ON or **OFF** displays or hides the cursor.

column and **row** specify the new position.

style defines a solid (1) or flashing (0) cursor.

Example: Using **CURSOR**

```
10  SCNCLR
20  CURSOR ON,1,2,1        :REM DISPLAY A SOLID CURSOR AT COLUMN 1, ROW 2
30  PRINT "A"; : SLEEP 1
40  CURSOR ,,0             :REM CHANGE TO A FLASHING CURSOR
50  PRINT "B"; : SLEEP 1
60  CURSOR OFF             :REM HIDE THE CURSOR
70  PRINT "C"; : SLEEP 1
80  CURSOR 20,10           :REM MOVE THE CURSOR TO COLUMN 20, ROW 10
90  PRINT "D"; : SLEEP 1
100 CURSOR ,50             :REM MOVE THE CURSOR TO ROW 5 BUT DO NOT CHANGE THE COLUMN
110 PRINT "E"; : SLEEP 1
100 CURSOR 0               :REM MOVE THE CURSOR TO THE START OF THE ROW
110 PRINT "F"; : SLEEP 1
```

DATA

Format: **DATA** [constant [**,** constant ...]]

Usage: Used to define constants which can be read by **READ** statements in a program. Numbers and strings are allowed, but expressions are not. Items are separated by commas. Strings containing commas, colons or spaces must be placed in quotes.

RUN initialises the data pointer to the first item of the first **DATA** statement and advances it for every read item. It is the programmer's responsibility that the type of the constant and the variable in the **READ** statement match. Empty items with no constant between commas are allowed and will be interpreted as zero for numeric variables and an empty string for string variables.

RESTORE may be used to set the data pointer to a specific line for subsequent reads.

Remarks: It is good programming practice to put large amounts of **DATA** statements at the end of the program, so they don't slow down the search for line numbers after **GOTO**, and other statements with line number targets.

Example: Using **DATA**

```
1 REM DATA
10 READ NA$, VE
20 READ N% : FOR I=2 TO N% : READ GL(I) : NEXT I
30 PRINT "PROGRAM:";NA$;"  VERSION:";VE
40 PRINT "N-POINT GAUSSLEGENDRE FACTORS E1":
50 FOR I=2 TO N%:PRINT I;GL(I):NEXT I
60 END
80 DATA "MEGA65",1.1
90 DATA 5,0.5120,0.3573,0.2760,0.2252

RUN
PROGRAM:MEGA65  VERSION: 1.1
N-POINT GAUSSLEGENDRE FACTORS E1
 2  0.512
 3  0.3573
 4  0.276
 5  0.2252
```

DCLEAR

Format: **DCLEAR** [**,D** drive] [**,U** unit]

Usage: Sends an initialise command to the specified unit and drive.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

The DOS of the disk drive will close all open files, clear all channels, free buffers and re-read the BAM. All open channels on the computer will also be closed.

Examples: Using **DCLEAR**

```
DCLEAR
DCLEAR U9
DCLEAR D0, U9
```

DCLOSE

Format: **DCLOSE** [**U** unit]
DCLOSE # channel

Usage: Closes a single file or all files for the specified unit.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

DCLOSE is used either with a channel argument or a unit number, but never both.

Remarks: It is important to close all open files before a program ends. Otherwise buffers will not be freed and even worse, open files that have been written to may be incomplete (commonly called splat files), and no longer usable.

Examples: Using **DCLOSE**

```
DCLOSE#2 :REM CLOSE FILE ASSIGNED TO CHANNEL 2
DCLOSE U9:REM CLOSE ALL FILES OPEN ON UNIT 9
```

DEC

Format: **DEC**(string expression)

Usage: Returns the decimal value of the argument, that is written as a hex string. The argument range is "0000" to "FFFF" (0 to 65535 in decimal). The argument must have 1-4 hex digits.

Remarks: Allowed digits in uppercase/graphics mode are 0-9 and A-Z (0123456789ABCDEF) and in lowercase/uppercase mode are 0-9 and a-z (0123456789abcdef).

Example: Using **DEC**

```
PRINT DEC("D000")
 53248
POKE DEC("600"),255
```

DEF FN

Format: **DEF FN** name(real variable) = [expression]

Usage: Defines a single statement user function with one argument of type real, returning a real value. The definition must be executed before the function can be used in expressions. The argument is a dummy variable, which will be replaced by the argument when the function is used.

Remarks: The value of the dummy variable will not change and the variable may be used in other contexts without side effects.

Example: Using **DEF FN**

```
10 PD = π / 180
20 DEF FN CD(X)= COS(X*PD): REM COS FOR DEGREES
30 DEF FN SD(X)= SIN(X*PD): REM SIN FOR DEGREES
40 FOR D=0 TO 360 STEP 90
50 PRINT USING "####";D
60 PRINT USING " ##.##";FNCD(D);
70 PRINT USING " ##.##";FNSD(D)
80 NEXT D
RUN
  0  1.00  0.00
 90  0.00  1.00
180 -1.00  0.00
270  0.00 -1.00
360  1.00  0.00
```

DELETE

Format: **DELETE** [line range]
DELETE filename [**,D** drive] [**,U** unit] [**,R**]

Usage: Used to either delete a range of lines from the BASIC program or to delete a file from disk.

line range consists of the first and last line to delete, or a single line number. If the first number is omitted, the first BASIC line is assumed. The second number in the range specifier defaults to the last BASIC line.

filename is either a quoted string, for example: `"SAFE""` or a string expression in brackets, for example: `(F$)`

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

R Recover a previously deleted file. This will only work if there were no write operations between deletion and recovery, which may have altered the contents of the file.

Remarks: **DELETE** is a synonym of **SCRATCH**.

Examples: Using **DELETE**

```
DELETE 100      :REM DELETE LINE 100
DELETE 240-350  :REM DELETE ALL LINES FROM 240 TO 350
DELETE 500-     :REM DELETE FROM 500 TO END
DELETE -70      :REM DELETE FROM START TO 70

DELETE "DRM",U9 :REM DELETE FILE DRM ON UNIT 9
```

DIM

Format: **DIM** name(limits) [**,** name(limits) ...]

Usage: Declares the shape, bounds and the type of a BASIC array. As a declaration statement, it must be executed only once and before any usage of the declared arrays. An array can have one or more dimensions. One dimensional arrays are often called vectors while two or more dimensions define a matrix. The lower bound of a dimension is always zero, while the upper bound is as declared. The rules for variable names apply for array names as well. You can create byte arrays, integer arrays, real arrays and string arrays. It is legal to use the same identifier for scalar variables and array variables. The left parenthesis after the name identifies array names.

Remarks: Byte arrays consume one byte per element, integer arrays two bytes, real arrays five bytes and string arrays three bytes for the string descriptor plus the length of the string itself.
If an array identifier is used without being previously declared, an implicit declaration of an one dimensional array with limit of 10 is performed.

Example: Using **DIM**

```
1 REM DIM
10 DIM A%(8)    : REM ARRAY OF 9 ELEMENTS
20 DIM XX(2,3)  : REM ARRAY OF 3X4 = 12 ELEMENTS
30 FOR I=0 TO 8 : A%(I)=PEEK(256+I) : PRINT A%(I);: NEXT:PRINT
40 FOR I=0 TO 2 : FOR J=0 TO 3 : READ XX(I,J):PRINT XX(I,J);: NEXT J,I
50 END
60 DATA 1,-2,3,-4,5,-6,7,-8,9,-10,11,-12

RUN
 45  52  50  0  0  0  0  0  0
 1 -2  3 -4  5 -6  7 -8  9 -10  11 -12
```

DIR

Format: **DIR** [filepattern] [**,W**] [**,R**] [**,D** drive] [**,U** unit]
DIRECTORY [filepattern] [**,W**] [**,R**] [**,D** drive] [**,U** unit]
$ [filepattern] [**,W**] [**,R**] [**,D** drive] [**,U** unit]

Usage: Prints a file directory/catalog of the specified disk.

The **W** (Wide) parameter lists the directory three columns wide on the screen and pauses after the screen has been filled with a page (63 directory entries). Pressing any key displays the next page.

The **R** (Recoverable) parameter includes files in the directory, which are flagged as deleted but are still recoverable.

filepattern is either a quoted string, for example: `"DA*"` or a string expression in brackets, e.g. `(DI$)`

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **DIR** is a synonym of **CATALOG** and **DIRECTORY**, and produces the same listing. The **filepattern** can be used to filter the listing. The wildcard characters * and **?** may be used. Adding **,T=** to the pattern string, with **T** specifying a filetype of **P**, **S**, **U** or **R** (for **P**RG, **S**EQ, **U**SR, **R**EL) filters the output to that filetype.

The shortcut symbol **$** can only be used in direct mode.

Examples: Using **DIR**

```
DIR
0 "BLACK SMURF    " BS 2A
508  "STORY PHOBOS"    SEQ
27   "C8096"           PRG
25   "C128"            PRG
104 BLOCKS FREE.
```

For a **DIR** listing with the **w**ide parameter, please refer to the example under **CATALOG** on page 89.

DISK

Format: **DISK** command [**,U** unit]
@ command [**,U** unit]

Usage: Sends a command string to the specified disk unit.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

command is a string expression.

Remarks: The command string is interpreted by the disk unit and must be compatible to the used DOS version. Read the disk drive manual for possible commands.

Using **DISK** with no parameters prints the disk status.

The shortcut key @ can only be used in direct mode.

Examples: Using **DISK**

```
DISK "I0"   :REM INITIALISE DISK IN DRIVE 0
DISK "U0>9" :REM CHANGE UNIT# TO 9
```

DLOAD

Format: **DLOAD** filename [**,D** drive] [**,U** unit]

Usage: "Disk LOAD" loads a file of type **PRG** into memory reserved for BASIC programs.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: The load address, which is stored in the first two bytes of the file is ignored. The program is loaded into BASIC memory. This enables loading of BASIC programs that were saved on other computers with different memory configurations. After loading, the program is re-linked and ready to be **RUN** or edited. It is possible to use **DLOAD** in a running program. This is called overlaying, or chaining. If you do this, then the newly loaded program replaces the current one, and the execution starts automatically on the first line of the new program. Variables, arrays and strings from the current run are preserved and can also be used by the newly loaded program.

Examples: Using **DLOAD**

```
DLOAD "APOCALYPSE"
DLOAD "MEGA TOOLS",U9
DLOAD (FI$),U(UN%)
```

DMA

Format: **DMA** command [**,** length**,** source address**,** source bank**,** target address**,** target bank [**,** sub]]

Usage: **DMA** ("Direct Memory Access") is obsolete, and has been replaced by **EDMA**.

command 0: copy, 1: mix, 2: swap, 3: fill

length number of bytes

source address 16-bit address of read area or fill byte

source bank bank number for source (ignored for fill mode)

target 16-bit address of write area

target bank bank number for target

sub sub command

Remarks: **DMA** has access to the lower 1MB address range organised in 16 banks of 64 K. To avoid this limitation, use **EDMA**, which has access to the full 256MB address range.

Examples: A sequence of **DMA** calls to demonstrate fast screen drawing operations

```
DMA 0, 80*25, 2048, 0, 0, 4  :REM SAVE SCREEN TO $00000 BANK 4
DMA 3, 80*25, 32, 0, 2048, 0 :REM FILL SCREEN WITH BLANKS
DMA 0, 80*25, 0, 4, 2048, 0  :REM RESTORE SCREEN FROM $00000 BANK 4
DMA 2, 80, 2048, 0, 2048+80, 0 :REM SWAP CONTENTS OF LINE 1 & 2 OF SCREEN
```

DMODE

Format: **DMODE** jam, complement, stencil, style, thick

Usage: "Display MODE" sets several parameters of the graphics context, which is used by drawing commands.

Mode	Values
jam	0 - 1
complement	0 - 1
stencil	0 - 1
style	0 - 3
thick	1 - 8

DO

Format: **DO** ... **LOOP**
DO [<**UNTIL** | **WHILE**> logical expression]
. . . statements [**EXIT**]
LOOP [<**UNTIL** | **WHILE**> logical expression]

Usage: **DO** and **LOOP** define the start of a BASIC loop. Using **DO** and **LOOP** alone without any modifiers creates an infinite loop, which can only be exited by the **EXIT** statement. The loop can be controlled by adding **UNTIL** or **WHILE** after the **DO** or **LOOP**.

Remarks: **DO** loops may be nested. An **EXIT** statement only exits the current loop.

Examples: Using **DO** and **LOOP**

```
10 PW$="":DO
20 GET A$:PW$=PW$+A$
30 LOOP UNTIL LEN(PW$))7 OR A$=CHR$(13)

10 DO : REM WAIT FOR USER DECISION
20 GET A$
30 LOOP UNTIL A$="Y" OR A$="N" OR A$="y" OR A$="n"

10 DO WHILE ABS(EPS) > 0.001
20 GOSUB 2000 : REM ITERATION SUBROUTINE
30 LOOP

10 I%=0 : REM INTEGER LOOP 1-100
20 DO I%=I%+1
30 LOOP WHILE I% < 101
```

DOPEN

Format: **DOPEN#** channel**,** filename [**,L** [reclen]] [**,W**] [**,D** drive] [**,U** unit]

Usage: Opens a file for reading or writing.

channel number, where:

- **1 <= channel <= 127** line terminator is CR.
- **128 <= channel <= 255** line terminator is CR LF.

L indicates, that the file is a relative file, which is opened for read-/write, as well as random access. The reclength is mandatory for creating relative files. For existing relative files, **reclen** is used as a safety check, if given.

W opens a file for write access. The file must not exist.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **DOPEN#** may be used to open all file types. The sequential file type **SEQ** is default. The relative file type **REL** is chosen by using the **L**

parameter. Other file types must be specified in the filename, e.g. by adding ",P" to the filename for **PRG** files or ",U" for **USR** files.

If the first character of the filename is an at sign '@', it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS.

Examples: Using **DOPEN**

```
DOPEN#5,"DATA",U9
DOPEN#130,(DD$),U(UN%)
DOPEN#3,"USER FILE,U"
DOPEN#2,"DATA BASE",L240
DOPEN#4,"MYPROG,P" : REM OPEN PRG FILE
```

DPAT

Format: **DPAT** type [**,** number**,** pattern ...]

Usage: "Drawing PATtern" sets the pattern of the graphics context for drawing commands.

There a four predefined pattern types, that can be selected by specifying the type number (1, 2, 3, or 4) as a single parameter.

A value of zero for the type number indicates a user defined pattern. This pattern can be set by using a bit string that consists of either 8, 16, 24, or 32 bits. The number of used pattern bytes is given as the second parameter. It defines how many pattern bytes (1, 2, 3, or 4) follow.

- **Type** 0-4
- **Number** number of following pattern bytes (1-4)
- **Pattern** pattern bytes

DS

Format: **DS**

Usage: **DS** holds the status of the last disk operation. It is a volatile variable. Each use triggers the reading of the disk status from the current disk device in usage. **DS** is coupled to the string variable **DS$** which is updated at the same time. Reading the disk status from a disk device automatically clears any error status on that device, so subsequent reads will return 0, if no other activity has since occurred.

Remarks: **DS** is a reserved system variable.

Example: Using **DS**

```
100 DOPEN#1,"DATA"
110 IF DS<>0 THEN PRINT"COULD NOT OPEN FILE DATA":STOP
```

DS$

Format: **DS$**

Usage: **DS$** holds the status of the last disk operation in text form of the format: Code,Message,Track,Sector.

DS$ is coupled to the numeric variable **DS**. It is updated when **DS** is used. DS$ is set to `00,OK,00,00` if there was no error, otherwise it is set to a DOS error message (listed in the disk drive manuals).

Remarks: **DS$** is a reserved system variable.

Example: Using **DS$**

```
100 DOPEN#1,"DATA"
110 IF DS<>0 THEN PRINT DS$:STOP
```

DSAVE

Format: **DSAVE** filename [**,D** drive] [**,U** unit]

Usage: "Disk SAVE" saves the BASIC program to a file of type **PRG**.

filename is either a quoted string such as `"DATA"`, or a string expression in brackets such as `(FI$)`. The maximum length of the filename is 16 characters. If the first character of the filename is an at sign '@' it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **DVERIFY** can be used after **DSAVE** to check if the saved program on disk is identical to the program in memory.

Example: Using **DSAVE**

```
DSAVE "ADVENTURE"
DSAVE "ZORK-I",U9
DSAVE "DUNGEON",D1,U10
```

DT$

Format: **DT$**

Usage: **DT$** holds the current date and is updated before each usage from the RTC (Real-Time Clock). The string **DT$** is formatted as: "DD-MON-YYYY", for example: "04-APR-2021".

Remarks: **DT$** is a reserved system variable. For more information on how to set the Real-Time Clock, refer to the Configuration Utility section on page 31.

Example: Using **DT$**

```
100 PRINT "TODAY IS: ";DT$
```

DVERIFY

Format: **DVERIFY** filename [**,D** drive] [**,U** unit]

Usage: "Disk VERIFY" compares the BASIC program in memory with a disk file of type **PRG**.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (F1$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **DVERIFY** can only test for equality. It gives no information about the number or position of different valued bytes. **DVERIFY** exits either with the message OK or with VERIFY ERROR.

Example: Using **DVERIFY**

```
DVERIFY "ADVENTURE"
DVERIFY "ZORK-I",U9
DVERIFY "DUNGEON",D1,U10
```

EDIT

Format: **EDIT** <**ON** | **OFF**>

Usage: **EDIT** switches the built-in editor either to text mode with **EDIT ON**, or to the BASIC program editor with**EDIT OFF**.

After power up or reset, the editor is initialised as BASIC program editor.

After setting the editor to text mode with **EDIT ON**, the differences to program mode are:

The editor does no tokenising/parsing. All text entered after a linenumber remains pure text, BASIC keywords such as **FOR** and **GOTO** are not converted to BASIC tokens, as they are whilst in program mode.

The line numbers are only used for text organisation, sorting, deleting, listing etc. When the text is saved to file with **DSAVE**, a sequential file (type **SEQ**) is written, not a program (**PRG**) file, which is how they're written whilst in program mode. Line numbers are not written to the file.

DLOAD in text mode can load only sequential files. Line numbers are automatically generated for editing purposes.

The mode of the editor can be recognised by looking at the prompt: In program mode, the prompt is `READY.`, whilst in text mode the prompt is `OK`.

Text mode affects entered lines with leading numbers only, lines with no line number are executed as BASIC commands, as usual.

Sequential files, created with the text editor, can be displayed (without loading them) on the screen by using **TYPE <filename>**.

Example: Using **EDIT**

```
ready.
edit on

ok.
100 This is a simple text editor.
dsave "example"

ok.
new

ok.
catalog

0 "demoempty      " 00 3d
1 "example"         seq
3159 blocks free

ok.
type "example"
This is a simple text editor.

ok.
dload "example"

loading

ok.
list

1000 This is a simple text editor.

ok.
```

EDMA

Format: **EDMA** command, length, source, target [, sub, mod]

Usage: **EDMA** ("Extended Direct Memory Access") is the fastest method to manipulate memory areas using the DMA controller.

command 0: copy, 1: mix, 2: swap, 3: fill.

length number of bytes (maximum = 65535).

source 28-bit address of read area or fill byte.

target 28-bit address of write area.

sub sub command (see chapter on DMA controller in the MEGA65 Book).

mod modifier (see chapter on DMA controller in the MEGA65 Book).

Remarks: **EDMA** can access the entire 256MB address range, using up to 28 bits for the addresses of the source and target.

Examples: Using **EDMA**

```
EDMA 0, $800, $F700, $8000000 :REM COPY SCALAR VARIABLES TO ATTIC RAM
EDMA 3, 80*25, 32, 2048       :REM FILL SCREEN WITH BLANKS
EDMA 0, 80*25, 2048, $8000800 :REM COPY SCREEN TO ATTIC RAM
```

EL

Format: **EL**

Usage: **EL** has the value of the line where the most recent BASIC error occurred, or the value -1 if there was no error.

Remarks: **EL** is a reserved system variable.

This variable is typically used in a **TRAP** routine, where the error line is taken from **EL**.

Example: Using **EL**

```
 10 TRAP 100
 20 PRINT SQR(-1)     :REM PROVOKE ERROR
 30 PRINT "AT LINE 30":REM HERE TO RESUME
 40 END
100 IF ER>0 THEN PRINT ERR$(ER);" ERROR"
110 PRINT " IN LINE";EL
120 RESUME NEXT       :REM RESUME AFTER ERROR
```

ELLIPSE

Format: **ELLIPSE** xcentre**,** ycentre**,** xradius**,** yradius [**,** solid]

Usage: Draws an ellipse.

xcentre x coordinate of centre in pixels.

ycentre y coordinate of centre in pixels.

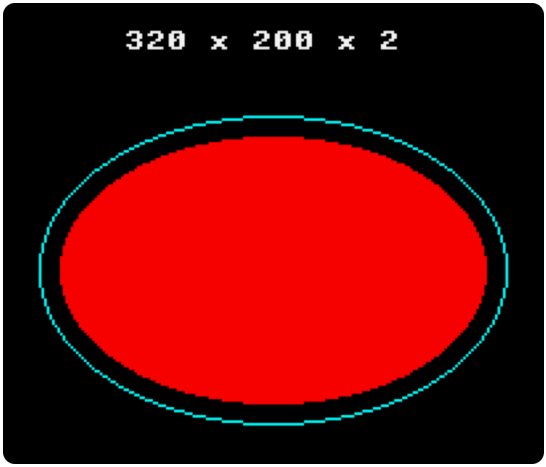
xradius x radius of the ellipse in pixels.

yradius y radius of the ellipse in pixels.

solid fills the ellipse, if not zero.

Remarks: **ELLIPSE** is used to draw ellipses on screens at various resolutions. It can also be used to draw circles.

Example: Using **ELLIPSE**




```
100 REM ELLIPSE
110 W=320:H=200:B%=2           :REM WIDTH, HEIGHT, DEPTH
120 X0=W/2:Y0=H/2:XD=W/4:YD=H/4 :REM CENTRE AND HALF AXIS
130 BORDER 0                   :REM BLACK
140 BACKGROUND 0               :REM BLACK
150 FOREGROUND 5               :REM GREEN
160 SCREEN 320,200,B%          :REM SET PARAMETERS
170 PEN 2                      :REM DRAW PEN RED
180 ELLIPSE X0,Y0,XD,YD,1      :REM DRAW SOLID ELLIPSE
190 PEN 3                      :REM DRAW PEN CYAN
200 ELLIPSE X0,Y0,XD+8,YD+8,0  :REM DRAW OUTLINED ELLIPSE
210 A$=STR$(W)+" X"+STR$(H)+" X"+STR$(B%)
220 PEN 1                      :REM DRAW PEN WHITE
230 CHAR  12,10,1,1,2,A$       :REM DRAW TEXT
240 GETKEY A$                  :REM WWAIT FOR KEYPRESS
250 SCREEN CLOSE               :REM CLOSE GRAPHICS
```

ELSE

Format: **IF** expression **THEN** true clause [**ELSE** false clause]

Usage: **ELSE** is an optional part of an **IF** statement.

expression a logical or numeric expression. A numeric expression is evaluated as **FALSE** if the value is zero and **TRUE** for any non-zero value.

true clause one or more statements starting directly after **THEN** on the same line. A line number after **THEN** performs a **GOTO** to that line instead.

false clause one or more statements starting directly after **ELSE** on the same line. A linenumber after **ELSE** performs a **GOTO** to that line instead.

Remarks: There must be a colon before **ELSE**. There cannot be a colon or end-of-line after **ELSE**.

The standard **IF ... THEN ... ELSE** structure is restricted to a single line. But the **true clause** and **false clause** may be expanded to several lines using a compound statement surrounded with **BEGIN** and **BEND**.

When the **true clause** does not use **BEGIN** and **BEND**, **ELSE** must be on the same line as **IF**.

Example: Using **ELSE**

```
100 REM ELSE
110 RED$=CHR$(28):BLACK$=CHR$(144):WHITE$=CHR$(5)
120 INPUT "ENTER A NUMBER";V
130 IF V<0 THENPRINT RED$;:ELSEPRINT BLACK$;
140 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
150 PRINT WHITE$
160 INPUT "END PROGRAM:(Y/N)";A$
170 IF A$="Y" THENEND
180 IF A$="N" THEN120:ELSE160
```

Using **ELSE** with **BEGIN** and **BEND**.

```
100 A = 0 : GOSUB 200
110 A = 1 : GOSUB 200
120 END
200 IF A = 0 THEN BEGIN
210 PRINT "HELLO"
220 BEND : ELSE BEGIN
230 PRINT "GOODBYE"
240 BEND
250 RETURN
```

END

Format: **END**

Usage: Ends the execution of the BASIC program. The READY. prompt appears and the computer goes into direct mode waiting for keyboard input.

Remarks: **END** does **not** clear channels nor close files. Also, variable definitions are still valid after **END**. The program may be continued with the **CONT** statement. After executing the last line of a program, **END** is automatically executed.

Example: Using **END**

```
10 IF V < 0 THEN END : REM NEGATIVE NUMBERS END THE PROGRAM
20 PRINT V
```

ENVELOPE

Format: **ENVELOPE** n [{**,** attack**,** decay**,** sustain**,** release**,** waveform**,** pw}]

Usage: Used to define the parameters for the synthesis of a musical instrument.

n envelope slot (0-9).

attack attack rate (0-15).

decay decay rate (0-15).

sustain sustain rate (0-15).

release release rate (0-15).

waveform 0: triangle, 1: sawtooth, 2: square/pulse, 3: noise, 4: ring modulation.

pw pulse width (0-4095) for waveform.

There are 10 slots for storing instrument parameters, preset with the following default values:

n	A	D	S	R	WF	PW	Instrument
0	0	9	0	0	2	1536	Piano
1	12	0	12	0	1		Accordion
2	0	0	15	0	0		Calliope
3	0	5	5	0	3		Drum
4	9	4	4	0	0		Flute
5	0	9	2	1	1		Guitar
6	0	9	0	0	2	512	Harpsichord
7	0	9	9	0	2	2048	Organ
8	8	9	4	1	2	512	Trumpet
9	0	9	0	0	0		Xylophone

Example: Using **ENVELOPE**

```
10 ENVELOPE 9,10,5,10,5,2,4000
20 VOL 9
30 TEMPO 30
40 PLAY "T904Q CDEFGAB U3T8 CDEFGAB L","T503Q H CGEQG T7 HCGEQG L"
```

ERASE

Format: **ERASE** filename [**,D** drive] [**,U** unit] [**,R**]

Usage: Used to erase a disk file.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

R Recover a previously erased file. This will only work if there were no write operations between erasing and recovery, which may have altered the contents of the file.

Remarks: **ERASE** is a synonym of **SCRATCH**.

The success and the number of erased files can be examined by printing or using the system variable **DS$**. The second to last number from **DS$** contains the number of successfully erased files, (normally the second to last number in **DS$** contains the track number in case of a disk error).

Examples: Using **ERASE**

```
ERASE "DRM",U9 :REM ERASE FILE DRM ON UNIT 9
PRINT DS$
01, FILES SCRATCHED,01,00
ERASE "OLD*"   :REM ERASE ALL FILES BEGINNING WITH "OLD"
PRINT DS$
01, FILES SCRATCHED,04,00
```

ER

Format: **ER**

Usage: **ER** has the value of the most recent BASIC error that has occurred, or -1 if there was no error.

Remarks: **ER** is a reserved system variable.

This variable is typically used in a **TRAP** routine, where the error number is taken from **ER**.

Example: Using **ER**

```
 10 TRAP 100
 20 PRINT SQR(-1)     :REM PROVOKE ERROR
 30 PRINT "AT LINE 30":REM HERE TO RESUME
 40 END
100 IF ER>0 THEN PRINT ERR$(ER);" ERROR"
110 PRINT " IN LINE";EL
120 RESUME NEXT       :REM RESUME AFTER ERROR
```

ERR$

Format: **ERR$**(number)

Usage: Used to convert an error number to an error string.

number is a BASIC error number (1-41).

This function is typically used in a **TRAP** routine, where the error number is taken from the reserved variable **ER**.

Remarks: Arguments out of range (1-41) will produce an ILLEGAL QUANTITY error.

Example: Using **ERR$**

```
 10 TRAP 100
 20 PRINT SQR(-1)     :REM PROVOKE ERROR
 30 PRINT "AT LINE 30":REM HERE TO RESUME
 40 END
100 IF ER>0 THEN PRINT ERR$(ER);" ERROR"
110 PRINT " IN LINE";EL
120 RESUME NEXT       :REM RESUME AFTER ERROR
```

EXIT

Format: **EXIT**

Usage: Exits the current **DO .. LOOP** and continues execution at the first statement after **LOOP**.

Remarks: In nested loops, **EXIT** exits only the current loop, and continues execution in an outer loop (if there is one).

Example: Using **EXIT**

```
1 REM EXIT
10 OPEN 2,8,0,"$"           : REM OPEN CATALOG
15 IF DS THEN PRINT DS$: STOP: REM CANT READ
20 GET#2,D$,D$              : REM DISCARD LOAD ADDRESS
25 DO                       : REM LINE LOOP
30   GET#2,D$,D$            : REM DISCARD LINE LINK
35   IF ST THEN EXIT        : REM END-OF-FILE
40   GET#2,LO,HI            : REM FILE SIZE BYTES
45   S=LO + 256 * HI        : REM FILE SIZE
50   LINE INPUT#2, F$       : REM FILE NAME
55   PRINT S;F$             : REM PRINT FILE ENTRY
60 LOOP
65 CLOSE 2
```

EXP

Format: **EXP**(numeric expression)

Usage: The **EXP** (EXPonential function) computes the value of the mathematical constant Euler's number (**2.71828183**) raised to the power of the argument.

Remarks: An argument greater than 88 produces an OVERFLOW ERROR.

Examples: Using **EXP**

```
PRINT EXP(1)
 2.71828183

PRINT EXP(0)
 1

PRINT EXP(LOG(2))
 2
```

FAST

Format: **FAST** [speed]

Usage: Set CPU clock to 1MHz, 3.5MHz or 40MHz.

speed CPU clock speed where:

- **1** sets CPU to 1MHz.
- **3** sets CPU to 3MHz.
- Anything other than **1** or **3** sets the CPU to 40MHz.

Remarks: Although it's possible to call **FAST** with any real number, the precision part (the decimal point and any digits after it), will be ignored.

FAST is a synonym of **SPEED**.

FAST has no effect if `POKE 0,65` has previously been used to set the CPU to 40MHz.

Example: Using **FAST**

```
10 FAST     :REM SET SPEED TO MAXIMUM (40 MHZ)
20 FAST 1   :REM SET SPEED TO 1 MHZ
30 FAST 3   :REM SET SPEED TO 3.5 MHZ
40 FAST 3.5 :REM SET SPEED TO 3.5 MHZ
```

FGOSUB

Format: **FGOSUB** numeric expression

Usage: Evaluates the given numeric expression, then calls (**GOSUB**s) the subroutine at the resulting line number.

Warning: Using this feature can break your program if **RENUMBER** is applied, as line numbers may change and the numeric expression will no longer address your intended line numbers.

Example: Using **FGOSUB**:

```
 10 INPUT "WHICH SUBROUTINE TO EXECUTE 100,200,300";LI
 20 FGOSUB LI     :REM HOPEFULLY THIS LINE # EXISTS
 30 GOTO 10       :REM REPEAT
100 PRINT "AT LINE 100":RETURN
200 PRINT "AT LINE 200":RETURN
300 PRINT "AT LINE 300":RETURN
```

FGOTO

Format: **FGOTO** numeric expression

Usage: Evaluates the given numeric expression, then jumps (**GO**es**TO**) to the resulting line number.

Warning: Using this feature can break your program if **RENUMBER** is applied, as line numbers may change and the numeric expression will no longer address your intended line numbers.

Example: Using **FGOTO**:

```
 10 INPUT "WHICH LINE # TO EXECUTE 100,200,300";LI
 20 FGOTO LI     :REM HOPEFULLY THIS LINE # EXISTS
 30 END
100 PRINT "AT LINE 100":END
200 PRINT "AT LINE 200":END
300 PRINT "AT LINE 300":END
```

FILTER

Format: **FILTER** sid [{**,** freq**,** lp**,** bp**,** hp**,** res}]

Usage: Sets the parameters for a SID sound filter.

sid 1: right SID, 2: left SID

freq filter cut off frequency (0 - 2047)

lp low pass filter (0: off, 1: on)

bp band pass filter (0: off, 1: on)

hp high pass filter (0: off, 1: on)

resonance resonance (0 - 15)

Remarks: Missing parameters keep their current value. The effective filter is the sum of of all filter settings. This enables band reject and notch effects.

Example: Using **FILTER**

```
10 PLAY "T7X103P9C"
15 SLEEP 0.02
20 PRINT "LOW PASS SWEEP" :L=1:B=0:H=0:GOSUB 100
30 PRINT "BAND PASS SWEEP":L=0:B=1:H=0:GOSUB 100
40 PRINT "HIGH PASS SWEEP":L=0:B=0:H=1:GOSUB 100
50 GOTO 20
100 REM *** SWEEP ***
110 FOR F = 50 TO 1950 STEP 50
120 IF F >= 1000 THEN FF = 2000-F : ELSE FF = F
130 FILTER 1,FF,L,B,H,15
140 PLAY "X1"
150 SLEEP 0.02
160 NEXT F
170 RETURN
```

FIND

Format: **FIND /**string**/** [**,** line range]
FIND "string**"** [**,** line range]

Usage: **FIND** is an editor command that can only be used in direct mode. It searches a given line range (if specified), otherwise the entire BASIC program is searched. At each occurrence of the "find string" the line is listed with the string highlighted. NO SCROLL can be used to pause the output.

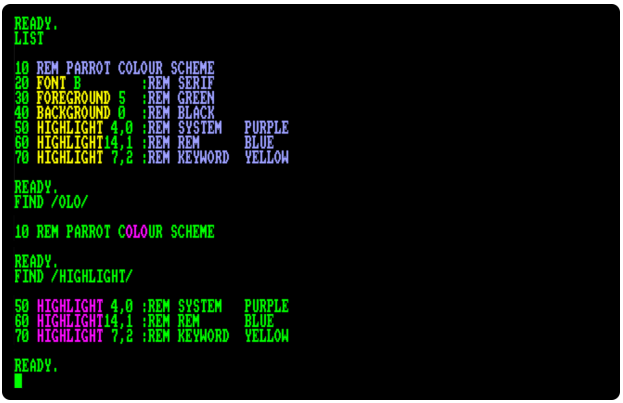
Remarks: Any un-shifted character that is not part of the string can be used instead of **/**.

However, using double quotes **"** as a delimiter has a special effect: The search text is not tokenised. **FIND "FOR"** will search for the three letters F, O, and R, not the BASIC keyword **FOR**. Therefore, it can find the word **FOR** in string constants or REM statements, but not in program code.

On the other hand, **FIND /FOR/** will find all occurrences of the BASIC keyword, but not the text "FOR" in strings.

Partial keywords cannot be searched. For example, **FIND /LOO/** will not find the keyword **LOOP**,

Example: Using **FIND**



FN

Format: **FN** name(numeric expression)

Usage: **FN** functions are user-defined functions, that accept a numeric expression as an argument and return a real value. They must first be defined with **DEF FN** before being used.

Example: Using **FN**

```
10 PD = π / 180
20 DEF FN CD(X)= COS(X*PD): REM COS FOR DEGREES
30 DEF FN SD(X)= SIN(X*PD): REM SIN FOR DEGREES
40 FOR D=0 TO 360 STEP 90
50 PRINT USING "###";D
60 PRINT USING " ##.##";FNCD(D);
70 PRINT USING " ##.##";FNSD(D)
80 NEXT D
RUN
  0  1.00  0.00
 90  0.00  1.00
180 -1.00  0.00
270  0.00 -1.00
360  1.00  0.00
```

FONT

Format: **FONT** <**A** | **B** | **C**>

Usage: **FONT** is used to switch between fonts, and the code pages PETSCII, and enhanced PETSCII. The enhanced PETSCII includes all ASCII sym-

bols that are missing in the PETSCII code page, although the order is still PETSCII. The ASCII symbols are typed by pressing the keys in the table below, some of which also require the holding down of the key. The codes for uppercase and lowercase are swapped compared to ASCII. The uppercase/graphics character set is not changed.

Code	Key	PETSCII	ASCII
$5C	Pound	£	\ (backslash)
$5E	Up Arrow (next to RESTORE)	↑	^ (caret)
$5F	Left Arrow (next to 1)	←	_ (underscore)
$7B	MEGA + Colon	┼	{ (open brace)
$7C	MEGA + Dot	▒	| (pipe)
$7D	MEGA + Semicolon	▏	} (close brace)
$7E	MEGA + Comma	▚	~ (tilde)

Remarks: The additional ASCII characters provided by FONT A and B are only available while using the lowercase/uppercase character set.

Examples: Using **FONT**

```
FONT A :REM ASCII - ENABLE {|}_~^
FONT B :REM LIKE A, WITH A SERIF FONT
FONT C :REM COMMODORE FONT (DEFAULT)
```

FOR

Format: **FOR** index = start **TO** end [**STEP** step] ... **NEXT** [index]

Usage: **FOR** statements start a BASIC loop with an index variable.

index may be incremented or decremented by a constant value on each iteration. The default is to increment the variable by 1. The index variable must be a real variable.

start is used to initialise the index.

end is checked at the end of an iteration, and determines whether another iteration will be performed, or if the loop will exit.

step defines the change applied to to the index variable at the end of an iteration. Positive step values increment it, while negative values decrement it. It defaults to 1.0 if not specified.

Remarks: For positive increments **end** must be greater than or equal to **start**, whereas for negative increments **end** must be less than or equal to **start**.

It is bad programming practice to change the value of the **index** variable inside the loop or to jump into or out of a loop body with **GOTO**.

Examples: Using **FOR**

```
10 FOR D=0 TO 360 STEP 30
20 R = D * π / 180
30 PRINT D;R;SIN(R);COS(R);TAN(R)
40 NEXT D

10 DIM M(20,20)
20 FOR I=0 TO 20
30 FOR J=I TO 20
40 M(I,J) = I + 100 * J
50 NEXT J,I
```

FOREGROUND

Format: **FOREGROUND** colour

Usage: Sets the foreground colour (text colour) of the screen to the argument, which must be in the range of 0 to 31. Refer to the table under **BACKGROUND** on page 79 for the colour values and their corresponding colours.

Remarks: **COLOR** also has the ability to change the foreground colour.

Example: Using **FOREGROUND**

```
READY.
FOREGROUND 7

READY.
```

FRE

Format: **FRE**(bank)

Usage: Returns the number of free bytes for banks 0 or 1, or the ROM version if the argument is negative.

FRE(0) returns the number of free bytes in bank 0, which is used for BASIC program source.

FRE(1) returns the number of free bytes in bank 1, which is the bank for BASIC variables, arrays and strings. **FRE(1)** also triggers "garbage

collection", which is a process that collects strings in use at the top of the bank, thereby defragmenting string memory.

FRE(-1) returns the ROM version, a six-digit number of the form `92XXXX`.

Example: Using **FRE**:

```
10 PM = FRE(0)
20 VM = FRE(1)
30 RV = FRE(-1)
40 PRINT PM;" FREE FOR PROGRAM"
50 PRINT VM;" FREE FOR VARIABLES"
60 PRINT RV;" ROM VERSION"
```

FREAD

Format: **FREAD#** channel**,** pointer**,** size

Usage: Reads **size** bytes from **channel** to memory starting at the 32-bit address **pointer**.

channel number, which was given to a previous call to commands such as **DOPEN**, or **OPEN**.

Care must be taken not to overwrite memory that is used by the system or the interpreter.

It is recommended to use the **POINTER** statement for the pointer argument, and to compute the size parameter by multiplying the number of elements with the item size.

Type	Item Size
Byte Array	1
Integer Array	2
Real Array	5

Keep in mind that the **POINTER** function with a string argument does NOT return the string address, but the string descriptor. It is not recommended to use **FREAD** for strings or string arrays unless you are fully aware on how to handle the string storage internals.

Also, ensure that you always specify an index if you use an array. The start address of array `XY()` is `POINTER(XY(0))`. `POINTER(XY)` returns the address of the scalar variable `XY`.

Example: Using **FREAD**:

```
100 N=23
110 DIM B&(N),C&(N)
120 DOPEN#2,"TEXT"
130 FREAD#2,POINTER(B&(0)),N
140 DCLOSE#2
150 FORI=0TON-1:PRINTCHR$(B&(I));:NEXT
160 FORI=0TON-1:C&(I)=B&(N-1-I):NEXT
170 DOPEN#2,"REVERS",W
180 FWRITE#2,POINTER(C&(0)),N
190 DCLOSE#2
```

FWRITE

Format: **FWRITE#** channel**,** pointer**,** size

Usage: Writes **size** bytes to **channel** from memory starting at the 32-bit address **pointer**.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**.

It is recommended to use the **POINTER** statement for the pointer argument and compute the size parameter by multiplying the number of elements with the item size.

Refer to the **FREAD** item size table on page 129 for the item sizes.

Keep in mind that the **POINTER** function with a string argument does NOT return the string address, but the string descriptor. It is not recommended to use **FWRITE** for strings or string arrays unless you are fully aware on how to handle the string storage internals.

Also, ensure that you always specify an index if you use an array. The start address of array `XY()` is `POINTER(XY(0))`. `POINTER(XY)` returns the address of the scalar variable `XY`.

Example: Using **FWRITE**:

```
100 N=23
110 DIM B&(N),C&(N)
120 DOPEN#2,"TEXT"
130 FREAD#2,POINTER(B&(0)),N
140 DCLOSE#2
150 FORI=0TON-1:PRINTCHR$(B&(I));:NEXT
160 FORI=0TON-1:C&(I)=B&(N-1-I):NEXT
170 DOPEN#2,"REVERS",W
180 FWRITE#2,POINTER(C&(0)),N
190 DCLOSE#2
```

GET

Format: **GET** variable

Usage: Gets the next character (or byte value of the next character) from the keyboard queue. If the variable being set to the character is of type string and the queue is empty, an empty string is assigned to it, otherwise a one character string is created and assigned instead. If the variable is of type numeric, the byte value of the key is assigned to it, otherwise zero will be assigned if the queue is empty. **GET** does not wait for keyboard input, so it's useful to check for key presses at regular intervals or in loops.

Remarks: **GETKEY** is similar, but waits until a key has been pressed.

Example: Using **GET**:

```
10 DO: GET A$: LOOP UNTIL A$ <> ""
40 IF A$ = "W" THEN 1000 :REM GO NORTH
50 IF A$ = "A" THEN 2000 :REM GO WEST
60 IF A$ = "S" THEN 3000 :REM GO EAST
70 IF A$ = "Z" THEN 4000 :REM GO SOUTH
80 IF A$ = CHR$(13) THEN 5000 :REM RETURN
90 GOTO 10
```

GET#

Format: **GET#** channel**,** variable [**,** variable ...]

Usage: Reads a single byte from the channel argument and assigns single character strings to string variables, or an 8-bit binary value to numeric variables. This is useful for reading characters (or bytes) from an input stream one byte at a time.

channel number, which was given to a previous call to commands such as **DOPEN**, or **OPEN**.

Remarks: All values from 0 to 255 are valid, so **GET** can also be used to read binary data.

Example: Using **GET#** to read a disk directory:

```
1 REM GET#
10 OPEN 2,8,0,"$"           : REM OPEN CATALOG
15 IF DS THEN PRINT DS$: STOP: REM CANT READ
20 GET#2,D$,D$              : REM DISCARD LOAD ADDRESS
25 DO                       : REM LINE LOOP
30   GET#2,D$,D$            : REM DISCARD LINE LINK
35   IF ST THEN EXIT        : REM END-OF-FILE
40   GET#2,LO,HI            : REM FILE SIZE BYTES
45   S=LO + 256 * HI        : REM FILE SIZE
50   LINE INPUT#2, F$       : REM FILE NAME
55   PRINT S;F$             : REM PRINT FILE ENTRY
60 LOOP
65 CLOSE 2
```

GETKEY

Format: **GETKEY** variable

Usage: Gets the next character (or byte value of the next character) from the keyboard queue. If the queue is empty, the program will wait until a key has been pressed. After a key has been pressed, the variable will be set and program execution will continue. When used with a string variable, a one character string is created and assigned. Otherwise if the variable is of type numeric, the byte value is assigned.

Example: Using **GETKEY**:

```
10 GETKEY A$ :REM WAIT AND GET CHARACTER
40 IF A$ = "W" THEN 1000 :REM GO NORTH
50 IF A$ = "A" THEN 2000 :REM GO WEST
60 IF A$ = "S" THEN 3000 :REM GO EAST
70 IF A$ = "Z" THEN 4000 :REM GO SOUTH
80 IF A$ = CHR$(13) THEN 5000 :REM RETURN
90 GOTO 10
```

GO64

Format: **GO64**

Usage: Switches the MEGA65 to C64-compatible -mode. If you're in direct mode, a security prompt `ARE YOU SURE?` is displayed, which must be responded with `Y` to continue. `SYS58552` can be used to switch back to C65-mode.

Example: Using **GO64**:

```
GO64
ARE YOU SURE?
```

GOSUB

Format: **GOSUB** line

Usage: **GOSUB** (GOto SUBroutine) continues program execution at the given BASIC line number, saving the current BASIC program counter and line number on the run-time stack. This enables the resumption of execution after the **GOSUB** statement, once a **RETURN** statement in the called subroutine is executed. Calls to subroutines via **GOSUB** may be nested, but the subroutines must always end with **RETURN**, otherwise a stack overflow may occur.

Remarks: Unlike other programming languages, BASIC 65 does not support arguments or local variables for subroutines.
Programs can be optimised by grouping subroutines at the beginning of the program source. The **GOSUB** calls will then have low line numbers with fewer digits to decode. The subroutines will also be found faster, since the search for subroutines often starts at the beginning of the program.

Example: Using **GOSUB**:

```
 10 GOTO 100 :REM TO MAIN PROGRAM
 20 REM *** SUBROUTINE DISK STATUS CHECK ***
 30 DD=DS:IF DD THEN PRINT "DISK ERROR";DS$
 40 RETURN
 50 REM *** SUBROUTINE PROMPT Y/N ***
 60 DO:INPUT "CONTINUE (Y/N)";A$
 70 LOOP UNTIL A$="Y" OR A$="N"
 80 RETURN
 90 REM *** MAIN PROGRAM ***
100 DOPEN#2,"BIG DATA"
110 GOSUB 30: IF DD THEN DCLOSE#2:GOSUB 60:REM ASK
120 IF A$="N" THEN STOP
130 GOTO 100: REM RETRY
```

GOTO

Format: **GOTO** line
GO TO line

Usage: Continues program execution at the given BASIC line number.

Remarks: If the target **line** number is higher than the current line number, the search starts from the current line, proceeding to higher line numbers. If the target **line** number is lower, the search starts at the first **line** number of the program. It is possible to optimise the run-time speed of the program by grouping often used targets at the start (with lower line numbers).

GOTO (written as a single word) executes faster than **GO TO**.

Example: Using **GOTO**:

```
 10 GOTO 100 :REM TO MAIN PROGRAM
 20 REM *** SUBROUTINE DISK STATUS CHECK ***
 30 DD=DS:IF DD THEN PRINT "DISK ERROR";DS$
 40 RETURN
 50 REM *** SUBROUTINE PROMPT Y/N ***
 60 DO:INPUT "CONTINUE (Y/N)";A$
 70 LOOP UNTIL A$="Y" OR A$="N"
 80 RETURN
 90 *** MAIN PROGRAM ***
100 DOPEN#2,"BIG DATA"
110 GOSUB 30: IF DD THEN DCLOSE#2:GOTO 60:REM ASK
120 IF A$="N" THEN STOP
130 GOTO 100: REM RETRY
```

GRAPHIC

Format: **GRAPHIC CLR**

Usage: Initialises the BASIC graphic system. It clears the graphics memory and screen, and sets all parameters of the graphics context to their default values.

Once the graphics system has been cleared, commands such as **LINE**, **PALETTE**, **PEN**, **SCNCLR**, and **SCREEN** can be used to set graphic system parameters again.

Example: Using **GRAPHIC**:

```
100 REM GRAPHIC
110 GRAPHIC CLR       : REM INITIALISE
120 SCREEN DEF 1,1,1,2 : REM 640 X 400 X 2
130 SCREEN OPEN 1     : REM OPEN IT
140 SCREEN SET 1,1    : REM VIEW IT
150 PALETTE 1,0,0, 0,0 : REM BLACK
160 PALETTE 1,1,0,15,0 : REM GREEN
170 SCNCLR 0          : REM FILL SCREEN WITH BLACK
180 PEN 0,1           : REM SELECT PEN
190 LINE 50,50,590,350 : REM DRAW LINE
200 GETKEY A$         : REM WAIT FOR KEYPRESS
210 SCREEN CLOSE 1    : REM CLOSE SCREEN AND RESTORE PALETTE
```

HEADER

Format: **HEADER** diskname [**,I** id] [**,D** drive] [**,U** unit]

Usage: Used to format (or clear) a disk.

I The disk ID.

diskname is either a quoted string, e.g. "DATA" or a string expression in brackets, e.g. (DN$). The maximum length of **diskname** is 16 characters.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: For new disks which have not already been formatted, it is necessary to specify the disk ID with the **I** parameter. This switches the format

command to full format, which writes sector IDs and erases all contents. This takes some time, as every block on the disk will be written. If the **I** parameter is omitted, a quick format will be performed. This is only possible if the disk has already been formatted. A quick format writes the new disk name and clears the block allocation map, marking all blocks as free. The disk ID is not changed, and blocks are not overwritten, so contents may be recovered with **ERASE R**. You can read more about **ERASE** on page 120.

Examples: Using **HEADER**

```
HEADER "ADVENTURE",IDK : FORMAT DISK WITH NAME ADVENTURE AND ID DK
HEADER "ZORK-I",U9     : FORMAT DISK IN UNIT 9 WITH NAME ZORK-I
HEADER "DUNGEON",D1,U10: FORMAT DISK IN DRIVE 1 UNIT 10 WITH NAME DUNGEON
```

HELP

Format: **HELP**

Usage: When the BASIC program stops due to an error, **HELP** can be used to gain further information. The interpreted line is listed, with the erroneous statement highlighted or underlined.

Remarks: Displays BASIC errors. For errors related to disk I/O, the disk status variable **DS** or the disk status string **DS$** should be used instead.

Example: Using **HELP**

```
10 A=1.E20
20 B=A+A:C=EXP(A):PRINT A,B,C
RUN

?OVERFLOW ERROR IN 20
READY.
HELP

20 B=A+A:C=EXP(A):PRINT A,B,C
```

HEX$

Format: **HEX$**(numeric expression)

Usage: Returns a four character hexadecimal representation of the argument. The argument must be in the range of 0-65535, corresponding to the hex numbers $0000-$FFFF.

Remarks: If real numbers are used as arguments, the fractional part will be ignored. In other words, real numbers will not be rounded.

Example: Using **HEX$**:

```
PRINT HEX$(10),HEX$(100),HEX$(1000.9)
000A      0064      03E8
```

HIGHLIGHT

Format: **HIGHLIGHT** colour [**,** mode]

Usage: Sets the colours used for highlighting. Different colours can be set for system messages, **REM** statements and BASIC 65 keywords.

colour is one of the first 16 colours in the current palette. Refer to page 79 for the colours in the default palette.

mode indicates what the colour will be used for.

- **0** system messages (the default mode)
- **1 REM** statements
- **2** BASIC keywords

Remarks: The system messages colour is used when displaying error messages, and in the output of **CHANGE**, **FIND**, and **HELP**. The colours for **REM** statements and BASIC keywords are used by **LIST**.

Example: Using **HIGHLIGHT** to change the color of BASIC keywords to red.

```
LIST

10 REM *** THIS IS HELLO WORLD ***
20 PRINT "HELLO WORLD"

READY.
HIGHLIGHT 8,2

READY.
LIST

10 REM *** THIS IS HELLO WORLD ***
20 PRINT "HELLO WORLD"

READY.
```

IF

Format: **IF** expression **THEN** true clause [**ELSE** false clause]

Usage: Starts a conditional execution statement.

expression a logical or numeric expression. A numeric expression is evaluated as **FALSE** if the value is zero and **TRUE** for any non-zero value.

true clause one or more statements starting directly after **THEN** on the same line. A line number after **THEN** performs a **GOTO** to that line instead.

false clause one or more statements starting directly after **ELSE** on the same line. A linenumber after **ELSE** performs a **GOTO** to that line instead.

Remarks: The standard **IF ... THEN ... ELSE** structure is restricted to a single line. But the **true clause** and **false clause** may be expanded to several lines using a compound statement surrounded with **BEGIN** and **BEND**.

Example: Using **IF**

```
1 REM IF
10 RED$=CHR$(28) : BLACK$=CHR$(144) : WHITE$=CHR$(5)
20 INPUT "ENTER A NUMBER";V
30 IF V<0 THEN PRINT RED$; : ELSE PRINT BLACK$;
40 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
50 PRINT WHITE$
60 INPUT "END PROGRAM: (Y/N)"; A$
70 IF A$="Y" THEN END
80 IF A$="N" THEN 20 : ELSE 60
```

INPUT

Format: **INPUT** [prompt <**,** | **;**>] variable [**,** variable ...]

Usage: Prints an optional prompt string and question mark to the screen, flashes the cursor and waits for user input from the keyboard.

prompt optional string expression to be printed as the prompt. If the separator between **prompt** and **variable list** is a comma, the cursor is placed directly after the prompt. If the separator is a semicolon, a question mark and a space is added to the prompt instead.

variable list list of one or more variables that receive the input.

The input will be processed after the user presses RETURN.

Remarks: The user must take care to enter the correct type of input, so it matches the **variable list** types. Also, the number of input items

must match the number of variables. A surplus of input items will be ignored, whereas too few input items trigger another request for input with the prompt ??. Typing non numeric characters for integer or real variables will produce a TYPE MISMATCH ERROR. Strings for string variables must be in double quotes (") if they contain spaces or commas. Many programs that need a safe input routine use **LINE INPUT** and a custom parser, in order to avoid program errors by wrong user input.

Example: Using **INPUT**:

```
 10 DIM N$(100),A%(100),S$(100):
 20 DO
 30 INPUT "NAME, AGE, GENDER";NA$,AG%,SE$
 40 IF NA$="" THEN 30
 50 IF NA$="END" THEN EXIT
 60 IF AG% < 18 OR AG% > 100 THEN PRINT "AGE?":GOTO 30
 70 IF SE$ <> "M" AND SE$ <> "F" THEN PRINT "GENDER?":GOTO 30
 80 REM CHECK OK: ENTER INTO ARRAY
 90 N$(N)=NA$:A%(N)=AG%:S$(N)=SE$:N=N+1
100 LOOP UNTIL N=100
110 PRINT "RECEIVED";N;" NAMES"
```

INPUT#

Format: **INPUT#** channel**,** variable [**,** variable ...]

Usage: Reads a record from an input device, e.g. a disk file and assigns the data to the variables in the list.

channel number, which was given to a previous call to commands such as **DOPEN**, or **OPEN**.

variable list list of one or more variables, that receive the input.

The input record must be terminated by a RETURN character and must be not longer than the input buffer (160 characters).

Remarks: The type and number of data in a record must match the variable list. Reading non numeric characters for integer or real variables will produce a FILE DATA ERROR. Strings for string variables have to be put in quotes if they contain spaces or commas.
LINE INPUT# may be used to read a whole record into a single string variable.

Sequential files, that can be read by **INPUT#** can be generated by programs with **PRINT#** or with the editor of the MEGA65. For example:

```
EDIT ON

10 "CHUCK PEDDLE",1937,"ENGINEER OF THE 6502"
20 "JACK TRAMIEL",1928,"FOUNDER OF CBM"
30 "BILL MENSCH",1945,"HARDWARE"

DSAVE "CBM-PEOPLE"
EDIT OFF
```

Example: Using **INPUT#**:

```
 10 DIM N$(100),B%(100),S$(100):
 20 DOPEN#2,"CBM-PEOPLE":REM OPEN SEQ FILE
 25 IF DS THEN PRINT DS$:STOP:REM OPEN ERROR
 30 FOR I=0 TO 100
 40 INPUT#2,N$(I),B%(I),S$(I)
 50 IF ST AND 64 THEN 80:REM END OF FILE
 60 IF DS THEN PRINT DS$:GOTO 80:REM DISK ERROR
 70 NEXT I
 80 DCLOSE#2
110 PRINT "READ";I;" RECORDS"
120 FOR J=0 TO I:PRINT N$(I):NEXT J

RUN
CHUCK PEDDLE
JACK TRAMIEL
BILL MENSCH

TYPE "CBM-PEOPLE"
"CHUCK PEDDLE",1937,"ENGINEER OF THE 6502"
"JACK TRAMIEL",1928,"FOUNDER OF CBM"
"BILL MENSCH",1945,"HARDWARE"
```

INSTR

Format: **INSTR**(haystack, needle [, start])

Usage: Locates the position of the string expression **needle** in the string expression **haystack**, and returns the index of the first occurrence, or zero if there is no match.

The string expression **haystack** is searched for the occurrence of the string expression **needle**.

An enhanced version of string search using pattern matching is used if the first character of the search string is a pound sign '£'. The pound sign is not part of the search but enables the use of the **'.'** (dot) as a wildcard character, which matches any character. The second special pattern character is the **'*'** (asterisk) character. The asterisk in the search string indicates that the preceding character may never appear, appear once, or repeatedly in order to be considered as a match.

The optional argument **start** is an integer expression, which defines the starting position for the search in **haystack**. If not present, it defaults to one.

Remarks: If either string is empty or there is no match the function returns zero.

Examples: Using **INSTR**:

```
I = INSTR("ABCDEF","CD")      : REM I = 3
I = INSTR("ABCDEF","XY")      : REM I = 0
I = INSTR("RAIIIN","£A*IN")   : REM I = 5
I = INSTR("ABCDEF","£C.E")    : REM I = 3
I = INSTR(A$+B$,C$)
```

INT

Format: **INT**(numeric expression)

Usage: Returns the integer part of the argument. This function is **NOT** limited to the typical 16-bit integer range (-32768 to 32767), as it uses real arithmetic. The allowed range is therefore determined by the size of the real mantissa which is 32-bits wide (-2147483648 to 2147483647).

Remarks: It is not necessary to use the **INT** function for assigning real values to integer variables, as this conversion will be done implicitly, but only for the 16-bit range.

Examples: Using **INT**:

```
X  = INT(1.9)      :REM X = 1
X  = INT(-3.1)     :REM X = -3
X  = INT(100000.5) :REM X = 100000
N% = INT(100000.5) :REM ?ILLEGAL QUANTITY ERROR
```

JOY

Format: **JOY**(port)

Usage: Returns the state of the joystick for the selected controller port (1 or 2). Bit 7 contains the state of the fire button. The stick can be moved in eight directions, which are numbered clockwise starting at the upper position.

	Left	Centre	Right
Up	8	1	2
Centre	7	0	3
Down	6	5	4

Example: Using **JOY**:

```
 10 N = JOY(1)
 20 IF N AND 128 THEN PRINT "FIRE! ";
 30 REM               N   NE  E   SE  S   SW  W   NW
 40 ON N AND 15 GOSUB 100,200,300,400,500,600,700,800
 50 GOTO 10
100 PRINT "GO NORTH"    :RETURN
200 PRINT "GO NORTHEAST":RETURN
300 PRINT "GO EAST"     :RETURN
400 PRINT "GO SOUTHEAST":RETURN
500 PRINT "GO SOUTH"    :RETURN
600 PRINT "GO SOUTHWEST":RETURN
700 PRINT "GO WEST"     :RETURN
800 PRINT "GO NORTHWEST":RETURN
```

KEY

Format: **KEY**
KEY <**ON** | **OFF**>
KEY <**LOAD** | **SAVE**> filename
KEY number**,** string

Usage: Reads the state of the function keys. The function keys can either send their key code when pressed, or a string assigned to the key. After power up or reset this feature is activated and the keys have their default assignments.

KEY list current assignments.

KEY ON switch on function key strings. The keys will send assigned strings if pressed.

KEY OFF switch off function key strings. The keys will send their character code if pressed.

KEY LOAD loads key definitions from file.

KEY SAVE saves key definitions to file.

KEY number, string assigns the string to the key with the given number.

Default assignments:

```
KEY
KEY 1,CHR$(27)+"X"
KEY 2,CHR$(27)+"C"
KEY 3,"DIR"+CHR$(13)
KEY 4,"DIR "+CHR$(34)+"*=PRG"+CHR$(34)+CHR$(13)
KEY 5,"▒"
KEY 6,"KEY6"+CHR$(141)
KEY 7,"▒"
KEY 8,"MONITOR"+CHR$(13)
KEY 9,"▒"
KEY 10,"KEY10"+CHR$(141)
KEY 11,"▒"
KEY 12,"KEY12"+CHR$(141)
KEY 13,CHR$(27)+"O"
KEY 14,"▒"+CHR$(27)+"O"
KEY 15,"HELP"+CHR$(13)
KEY 16,"RUN "+CHR$(34)+"*"+CHR$(34)+CHR$(13)
```

Remarks: The sum of the lengths of all assigned strings must not exceed 240 characters. Special characters such as RETURN or QUOTE are entered using their codes with the **CHR$** function. Refer to **CHR$** on page 92 for more information.

Examples: Using **KEY**:

```
KEY ON                 :REM ENABLE  FUNCTION KEYS
KEY OFF                :REM DISABLE FUNCTION KEYS
KEY                    :REM LIST ASSIGNMENTS
KEY 2,"PRINT π"+CHR$(14) :REM ASSIGN PRINT PI TO F2
KEY SAVE "MY KEY SET"  :REM SAVE CURRENT DEFINITIONS TO FILE
KEY LOAD "ELEVEN-SET"  :REM LOAD DEFINITIONS FROM FILE
```

LEFT$

Format: **LEFT$**(string**,** n)

Usage: Returns a string containing the first **n** characters from the argument **string**. If the length of **string** is equal to or less than **n**, the resulting string will be identical to the argument string.

string a string expression.

n a numeric expression (0-255).

Remarks: Empty strings and zero length strings are legal values.

Example: Using **LEFT$**:

```
PRINT LEFT$("MEGA-65",4)
MEGA
```

LEN

Format: **LEN**(string)

Usage: Returns the length of a string.

string a string expression.

Remarks: There is no terminating character, as opposed to other programming languages such as C, which uses the NULL character. The length of the string is internally stored in an extra byte of the string descriptor.

Example: Using **LEN**:

```
PRINT LEN("MEGA-65"+CHR$(13))
 8
```

LET

Format: [**LET**] variable = expression

Usage: Assigns values (or results of expressions) to variables.

Remarks: The **LET** statement is obsolete and not required. Assignment to variables can be done without using **LET**, but it has been left in BASIC 65 for backwards compatibility.

Examples: Using **LET**:

```
LET A=5  :REM LONGER  AND SLOWER
A=5      :REM SHORTER AND FASTER
```

LINE

Format: **LINE** xbeg**,** ybeg [**,** xnext1**,** ynext1 ...]

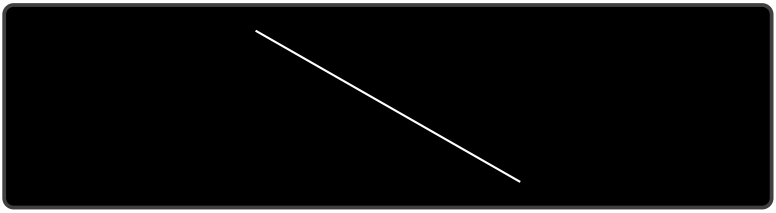
Usage: Draws a pixel at (xbeg/ybeg), if only one coordinate pair is given.

If more than one pair is defined, a line is drawn on the current graphics screen from the coordinate (xbeg/ybeg) to the next coordinate pair(s).

All currently defined modes and values of the graphics context are used.

Example: Using **LINE**:

```
1 REM SCREEN EXAMPLE 1
10 SCREEN 320,200,2      :REM SCREEN #0 320 X 200 X 2
20 PEN 1                 :REM DRAWING PEN COLOR 1 (WHITE)
30 LINE 25,25,295,175    :REM DRAW LINE
40 GETKEY A$             :REM WAIT FOR KEYPRESS
50 SCREEN CLOSE          :REM CLOSE SCREEN AND RESTORE PALETTE
```



LINE INPUT#

Format: **LINE INPUT#** channel**,** variable [**,** variable ...]

Usage: Reads one record per variable from an input device, (such as a disk drive) and assigns the read data to the variable. The records must be terminated by a **RETURN** character, which will not be copied to the string variable. Therefore, an empty line consisting of only the **RETURN** character will result in an empty string being assigned.

channel number, which was given to a previous call to commands such as **DOPEN**, or **OPEN**.

variable list list of one or more variables, that receive the input.

Remarks: Only string variables or string array elements can be used in the variable list. Unlike other INPUT commands, **LINE INPUT#** does not interpret or remove quote characters in the input. They are accepted as data, as all other characters.

Records must not be longer than the input buffer, which is 160 characters.

Example: Using **LINE INPUT#**:

```
 10 DIM N$(100)
 20 DOPEN#2,"DATA"
 30 FOR I=0 TO 100
 40 LINE INPUT#2,N$(I)
 50 IF ST=64 THEN 80:REM END OF FILE
 60 IF DS THEN PRINT DS$:GOTO 80:REM DISK ERROR
 70 NEXT I
 80 DCLOSE#2
110 PRINT "READ";I;" RECORDS"
```

LIST

Format: **LIST** [**P**] [line range]

Usage: Used to list a range of lines from the BASIC program.

line range consists of the first and/or last line to list, or a single line number. If the first number is omitted, the first BASIC line is assumed. If the second number is omitted, the last BASIC line is assumed.

Format: **LIST** [**P**] filename [**,U** unit]

Usage: Used to list a BASIC program directly from **unit**, which by default is 8.

Remarks: The optional parameter **P** enables page mode. After listing 24 lines, the listing will stop and display the prompt [MORE] at the bottom of the screen. Pressing **Q** quits page mode, while any other key triggers the listing of the next page.

LIST output can be redirected to other devices via **CMD**.

The keys F9 and F11, or Ctrl P and Ctrl V scroll a BASIC listing on screen up or down.

Examples: Using **LIST**

```
LIST 100      :REM LIST LINE 100
LIST 240-350  :REM LIST ALL LINES FROM 240 TO 350
LIST 500-     :REM LIST FROM 500 TO END
LIST -70      :REM LIST FROM START TO 70
LIST "DEMO"   :REM LIST FILE "DEMO"
LIST P        :REM LIST PROGRAM IN PAGE MODE
LIST P "MURX" :REM LIST FILE "MURX" IN PAGE MODE
```

LOAD

Format: **LOAD** filename [**,** unit [**,** flag]]
/ filename [**,** unit [**,** flag]]

Usage: A common use of the shortcut symbol **/** is to quickly load **PRG** files. To do this:

1. Print a disk directory using either **DIR**, or **CATALOG**.
2. Move the cursor to the desired line.
3. type **/** in the first column of the line, and press RETURN.

After pressing RETURN, the listed file on the line with the leading **/** will be loaded. Characters before and after the file name double quotes (") will be ignored. This applies to **PRG** files only.

filename is either a quoted string, e.g. "PROG", or a string expression.

The unit number is optional. If not present, the default disk device is assumed.

If **flag** has a non-zero value, the file is loaded to the address which is read from the first two bytes of the file. Otherwise, it is loaded to the start of BASIC memory and the load address in the file is ignored.

Remarks: **LOAD** loads files of type **PRG** into RAM bank 0, which is also used for BASIC program source.

LOAD "*" can be used to load the first **PRG** from the given **unit**.

LOAD "$" can be be used to load the list of files from the given **unit**. When using **LOAD "$"**, **LIST** can be used to print the listing to screen.

LOAD is implemented in BASIC 65 to keep it backwards compatible with BASIC V2.

The shortcut symbol **/** can only be used in direct mode.

By default the C64 uses **unit** 1, which is assigned to datasette tape recorders connected to the cassette port. However the MEGA65

uses **unit** 8 by default, which is assigned to the internal disk drive. This means you don't need to add ,8 to **LOAD** commands that use it.

Examples: Using **LOAD**

```
LOAD "APOCALYPSE"   :REM LOAD A FILE CALLED APOCALYPSE TO BASIC MEMORY
LOAD "MEGA TOOLS",9 :REM LOAD A FILE CALLED "MEGA TOOLS" FROM UNIT 9 TO BASIC MEMORY
LOAD "*",8,1        :LOAD THE FIRST FILE ON UNIT 8 TO RAM AS SPECIFIED IN THE FILE
```

LOADIFF

Format: **LOADIFF** filename [**,D** drive] [**,U** unit]

Usage: Loads an IFF file into graphics memory. The IFF (Interchange File Format) is supported by many different applications and operating systems. **LOADIFF** assumes that files contain bitplane graphics which fit into the MEGA65 graphics memory. Supported resolutions are:

Width	Height	Bitplanes	Colours	Memory
320	200	max. 8	max. 256	max. 64 K
640	200	max. 8	max. 256	max. 128 K
320	400	max. 8	max. 256	max. 128 K
640	400	max. 4	max. 16	max. 128 K

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: Tools are available to convert popular image formats to IFF. These tools are available on several operating systems, such as AMIGA OS, macOS, Linux, and Windows. For example, **ImageMagick** is a free graphics package that includes a tool called **convert**, which can be used to create IFF files in conjunction with the **ppmtoilbm** tool from the **Netbpm** package.

To use **convert** and **ppmtoilbm** for converting a JPG file to an IFF file on Linux:

```
convert <myImage.jpg> <myImage.ppm>
ppmtoilbm -aga <myImage.ppm> > <myImage.iff>
```

Example: Using **LOADIFF**

```
100 BANK128:SCNCLR
110 REM DISPLAY PICTURES IN 320 X 200 X 7 RESOLUTION
120 GRAPHIC CLR:SCREEN DEF 0,0,0,7:SCREEN OPEN 0:SCREEN SET 0,0
130 FORI=1TO7: READF$
140 LOADIFF(F$+".IFF"):SLEEP 4:NEXT
150 DATA ALIEN,BEAKER,JOKER,PICARD,PULP,TROOPER,RIPLEY
160 SCREEN CLOSE 0
170 PALETTE RESTORE
```

LOG

Format: **LOG**(numeric expression)

Usage: Computes the value of the natural logarithm of the argument. The natural logarithm uses Euler's number (**2.71828183**) as base, not 10 which is typically used in log functions on a pocket calculator.

Remarks: The log function with base 10 can be computed by dividing the result by log(10).

Example: Using **LOG**

```
PRINT LOG(1)
 0

PRINT LOG(0)
 ?ILLEGAL QUANTITY ERROR

PRINT LOG(4)
 1.38629436

PRINT LOG(100) / LOG(10)
 2
```

LOG10

Format: **LOG10**(numeric expression)

Usage: Computes the value of the decimal logarithm of the argument. The decimal logarithm uses 10 as base.

Example: Using **LOG10**

```
PRINT LOG10(1)
 0

PRINT LOG10(0)
 ?ILLEGAL QUANTITY ERROR

PRINT LOG10(5)
 0.69897

PRINT LOG10(100);LOG(10);LOG(0.1);LOG(0.01)
 2  1 -1 -2
```

LOOP

Format: **DO** ... **LOOP**
DO [<**UNTIL** | **WHILE**> logical expression]
. . . statements [**EXIT**]
LOOP [<**UNTIL** | **WHILE**> logical expression]

Usage: **DO** and **LOOP** define the start of a BASIC loop. Using **DO** and **LOOP** alone without any modifiers creates an infinite loop, which can only be exited by the **EXIT** statement. The loop can be controlled by adding **UNTIL** or **WHILE** after the **DO** or **LOOP**.

Remarks: **DO** loops may be nested. An **EXIT** statement only exits the current loop.

Examples: Using **DO** and **LOOP**

```
10 PW$="":DO
20 GET A$:PW$=PW$+A$
30 LOOP UNTIL LEN(PW$)>7 OR A$=CHR$(13)

10 DO : REM WAIT FOR USER DECISION
20 GET A$
30 LOOP UNTIL A$="Y" OR A$="N" OR A$="y" OR A$="n"

10 DO WHILE ABS(EPS) > 0.001
20 GOSUB 2000 : REM ITERATION SUBROUTINE
30 LOOP

10 I%=0 : REM INTEGER LOOP 1-100
20 DO I%=I%+1
30 LOOP WHILE I% < 101
```

LPEN

Format: **LPEN**(coordinate)

Usage: This function requires the use of a CRT monitor (or TV), and a light pen. It will not work with an LCD or LED screen. The light pen must be connected to port 1.

LPEN(0) returns the X position of the light pen, the range is 60-320.

LPEN(1) returns the Y position of the light pen, the range is 50-250.

Remarks: The X resolution is two pixels, therefore **LPEN(0)** only returns even numbers. A bright background colour is needed to trigger the light pen. The **COLLISION** statement may be used to enable an interrupt handler.

Example: Using **LPEN**

```
PRINT LPEN(0),LPEN(1)   :REM PRINT LIGHT PEN COORDINATES
```

MERGE

Format: **MERGE** filename [**,D** drive] [**,U** unit]

Usage: **MERGE** loads a BASIC program file from disk and appends it to the program in memory.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: The load address, which is stored in the first two bytes of the file is ignored. The loaded program does not replace a program in memory (which is what **DLOAD** does), but is appended to a program in memory. After loading, the program is re-linked and ready to run or edit.

It is the user's responsibility to ensure that there are no line number conflicts among the program in memory and the merged program. The first line number of the merged program must be greater than the last line number of the program in memory.

Example: Using **MERGE**

```
DLOAD "MAIN PROGRAM"
MERGE "LIBRARY"
```

MID$

Format: **MID$**(string**,** index**,** n)
MID$(string**,** index**,** n) = string expression

Usage: **MID$** can be used either as a function which returns a string, or as a statement for inserting sub-strings into an existing string.

string a string expression.

index start index (0-255).

n length of sub-string (0-255).

Remarks: Empty strings and zero lengths are legal values.

Example: Using **MID$**:

```
10 A$ = "MEGA-65"
20 PRINT MID$(A$,3,4)
30 MID$(A$,5,1) = "+"
40 PRINT A$
RUN
GA-6
MEGA+65
```

MOD

Format: **MOD**(dividend**,** divisor)

Usage: The **MOD** function returns the remainder of the division.

Remarks: In other programming languages such as C, this function is implemented as an operator (%). In BASIC 65 it is implemented as a function.

Example: Using **MOD**:

```
FOR I = 0 TO 8: PRINT MOD(I,4);: NEXT I
 0  1  2  3  0  1  2  3  0
```

MONITOR

Format: **MONITOR**

Usage: Calls the machine language monitor program, which is mainly used for debugging.

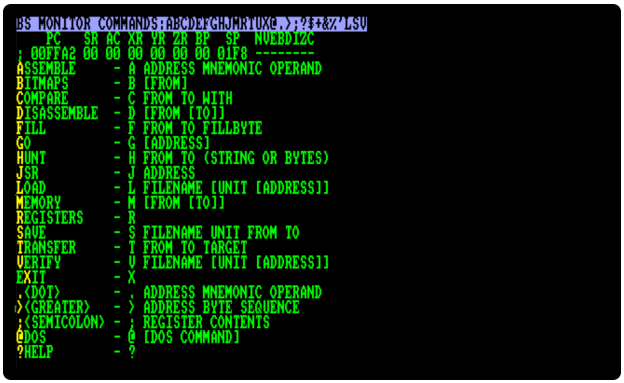
Remarks: Using the **MONITOR** requires knowledge of the CSG4510 / 6502 / 6510 CPU, the assembly language they use, and their architectures. More information on the **MONITOR** is available in the **MEGA65 Book**.

To exit the monitor press **X**.

Help text can be displayed with either **?** or **H**.

Example: Using **MONITOR**

```
MONITOR
```


BS MONITOR COMMANDS:ABCDEFGHJLMRTUXG >1#+#%'LSU
PC SR AC XR YR ZR BP SP NVEBDIZC
; 00FFA2 00 00 00 00 00 00 01F8 --------
ASSEMBLE - A ADDRESS MNEMONIC OPERAND
BITMAPS - B [FROM]
COMPARE - C FROM TO WITH
DISASSEMBLE - D [FROM [TO]]
FILL - F FROM TO FILLBYTE
GO - G [ADDRESS]
HUNT - H FROM TO (STRING OR BYTES)
JSR - J ADDRESS
LOAD - L FILENAME [UNIT [ADDRESS]]
MEMORY - M [FROM [TO]]
REGISTERS - R
SAVE - S FILENAME UNIT FROM TO
TRANSFER - T FROM TO TARGET
VERIFY - V FILENAME [UNIT [ADDRESS]]
EXIT - X
.<DOT> - . ADDRESS MNEMONIC OPERAND
)<GREATER> -) ADDRESS BYTE SEQUENCE
;<SEMICOLON> - ; REGISTER CONTENTS
@DOS - @ [DOS COMMAND]
?HELP - ?

MOUSE

Format: **MOUSE ON** [{, port, sprite, pos}]
MOUSE OFF

Usage: Enables the mouse driver and connects the mouse at the specified port with the mouse pointer sprite.

port mouse port 1, 2 (default) or 3 (both).

sprite sprite number for mouse pointer (default 0).

pos initial mouse position (x,y).

MOUSE OFF disables the mouse driver and frees the associated sprite.

Remarks: The "hot spot" of the mouse pointer is the upper left pixel of the sprite.

Examples: Using **MOUSE**:

```
REM LOAD DATA INTO SPRITE #0 BEFORE USING IT
MOUSE ON, 1       :REM ENABLE  MOUSE WITH SPRITE #0
MOUSE OFF         :REM DISABLE MOUSE
```

MOVSPR

Format: **MOVSPR** number**,** position

Usage: Moves a sprite on screen. Each **position** argument consists of two 16-bit values, which specify either an absolute coordinate, a relative coordinate, an angle, or a speed. The value type is determined by a prefix:

- **+value** relative coordinate: positive offset.
- **-value** relative coordinate: negative offset.
- **#value** speed.

If no prefix is given, the absolute coordinate or angle is used.

Therefore, the position argument can be used to either:

- set the sprite to an absolute position on screen.
- specify a displacement relative from the current position.
- trigger a relative movement from a specified position.
- describe movement with an angle and speed starting from the current position.

MOVSPR number, position is used to set the sprite immediately to the position or, in the case of an angle#speed argument, describe its further movement.

Format: **MOVSPR** number**,** start-position **TO** end-position**,** speed

Usage: Places the sprite at the start position, defines the destination position, and the speed of movement. The sprite is placed at the start position, and will move in a straight line to the destination at the given speed. Coordinates must be absolute or relative. The movement is controlled by the BASIC interrupt handler and happens concurrently with the program execution.

number sprite number (0-7).

position x,y | xrel,y | x,yrel | xrel,yrel | angle#speed.

x absolute screen coordinate pixel.

y absolute screen coordinate pixel.

xrel relative screen coordinate pixel.

yrel relative screen coordinate pixel.

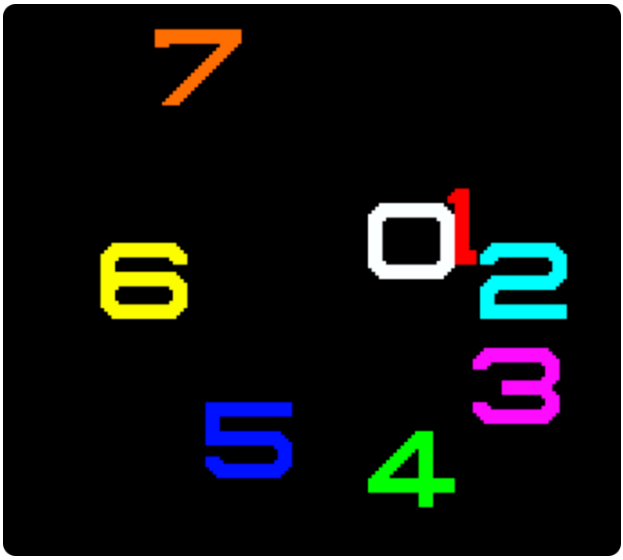
angle compass direction for sprite movement [degrees]. 0: up, 90: right, 180: down, 270: left, 45 upper right, etc.

speed speed of movement, configured as a floating point number in the range of 0.0–127.0, in pixels per frame. PAL has 50 frames per second whereas NTSC has 60 frames per second. A speed value of 1.0 will move the sprite 50 pixels per second in PAL mode.

Remarks: The "hot spot" is the upper left pixel of the sprite.

Example: Using **MOVSPR**:

```
100 CLR:SCNCLR:SPRITECLR
110 BLOAD "DEMOSPRITES1",B0,P1536
130 FORI=0TO7: C=I+1:SP=0.07*(I+1)
140 MOVSPRI, 160,120
145 MOVSPRI,45*I#SP
150 SPRITEI,1,C,,0,0
160 NEXT
170 SLEEP 3
180 FORI=0TO7:MOVSPR I,0#0:NEXT
```



NEW

Format: **NEW**
NEW RESTORE

Usage: Resets all BASIC parameters to their default values. Since **NEW** resets parameters and pointers, (but does not overwrite the address range of a BASIC program that was in memory), it is possible to recover the program. If there were no **LOAD** operations, or editing performed after **NEW**, the program can be restored with the **NEW RESTORE**.

Examples: Using **NEW**:

```
NEW         :REM RESET BASIC
NEW RESTORE :REM TRY TO RECOVER NEW'ED PROGRAM
```

NEXT

Format: **FOR** index = start **TO** end [**STEP** step] ... **NEXT** [index]

Usage: Marks the end of the BASIC loop associated with the given index variable. When a BASIC loop is declared with **FOR**, it must end with **NEXT**.

The **index** variable may be incremented or decremented by a constant value **step** on each iteration. The default is to increment the variable by 1. The index variable must be a real variable.

start value to initialise the index with.

end is checked at the end of an iteration, and determines whether another iteration will be performed, or if the loop will exit.

step defines the change applied to to the index variable at the end of every iteration. Positive step values increment it, while negative values decrement it. It defaults to 1.0 if not specified.

Remarks: The **index** variable after **NEXT** is optional. If it is omitted, the variable for the current loop is assumed. Several consecutive **NEXT** statements may be combined by specifying the indexes in a comma separated list. The statements `NEXT I:NEXT J:NEXT K` and `NEXT I,J,K` are equivalent.

Example: Using **NEXT**

```
10 FOR D=0 TO 360 STEP 30
20 R = D * π / 180
30 PRINT D;R;SIN(R);COS(R);TAN(R)
40 NEXT D

10 DIM M(20,20)
20 FOR I=0 TO 20
30 FOR J=I TO 20
40 M(I,J) = I + 100 * J
50 NEXT J,I
```

NOT

Format: **NOT** operand

Usage: Performs a bit-wise logical NOT operation on a 16-bit value. Integer operands are used as they are, whereas real operands are converted to a signed 16-bit integer (losing precision). Logical operands are converted to a 16-bit integer, using $FFFF (decimal -1) for TRUE, and $0000 (decimal 0) for FALSE.

Expression	Result
NOT 0	1
NOT 1	0

Remarks: The result is of type integer.

Examples: Using **NOT**

```
PRINT NOT 3
-4
PRINT NOT 64
-65
```

In most cases, **NOT** is used in **IF** statements.

```
OK = C < 256 AND C >= 0
IF (NOT OK) THEN PRINT "NOT A BYTE VALUE"
```

OFF

Format: keyword **OFF**

Usage: **OFF** is a secondary keyword used in combination with primary keywords, such as **COLOR**, **KEY**, and **MOUSE**.

Remarks: **OFF** cannot be used on its own.

Examples: Using **OFF**

```
COLOR OFF :REM DISABLE SCREEN COLOUR
KEY OFF   :REM DISABLE FUNCTION KEY STRINGS
MOUSE OFF :REM DISABLE MOUSE DRIVER
```

ON

Format: **ON** expression **GOSUB** line number [**,** line number ...]
ON expression **GOTO** line number [**,** line number ...]
keyword **ON**

Usage: **ON** calls either a computed **GOSUB** or **GOTO** statement. Depending on the result of the expression, the target for **GOSUB** and **GOTO** is chosen from the table of line addresses at the end of the statement.

When used as a secondary keyword, **ON** is used in combination with primary keywords, such as **COLOR**, **KEY**, and **MOUSE**.

expression is a positive numeric value. Real values are converted to integer (losing precision). Logical operands are converted to a 16-bit integer, using $FFFF (decimal -1) for TRUE, and $0000 (decimal 0) for FALSE.

Remarks: Negative values for **expression** will stop the program with an error message. The **line number list** specifies the targets for values of 1, 2, 3, etc.
An expression result of zero, or a result that is greater than the number of target lines will not do anything, and the program will continue execution with the next statement.

Example: Using **ON**

```
 10 COLOR ON :REM ENABLE SCREEN COLOUR
 20 KEY   ON :REM ENABLE FUNCTION KEY STRINGS
 30 MOUSE ON :REM ENABLE MOUSE DRIVER
 40 N = JOY(1):IF N AND 128 THEN PRINT "FIRE! ";
 60 REM              N   NE  E   SE  S   SW  W   NW
 70 ON N AND 15 GOSUB 100,200,300,400,500,600,700,800
 80 GOTO 40
100 PRINT "GO NORTH"    :RETURN
200 PRINT "GO NORTHEAST":RETURN
300 PRINT "GO EAST"     :RETURN
400 PRINT "GO SOUTHEAST":RETURN
500 PRINT "GO SOUTH"    :RETURN
600 PRINT "GO SOUTHWEST":RETURN
700 PRINT "GO WEST"     :RETURN
800 PRINT "GO NORTHWEST":RETURN
```

OPEN

Format: **OPEN** channel**,** first address [**,** secondary address [**,** filename]]

Usage: Opens an input/output channel for a device.

channel number, where:

- **1 <= channel <= 127** line terminator is CR.
- **128 <= channel <= 255** line terminator is CR LF.

first address device number. For IEC devices the unit number is the primary address. Following primary address values are possible:

Unit	Device
0	Keyboard
1	System Default
2	RS232 Serial Connection
3	Screen
4-7	IEC Printer and Plotter
8-31	IEC Disk Drives

The **secondary address** has some reserved values for IEC disk units, 0: load, 1: save, 15: command channel. The values 2-14 may be used for disk files.

filename is either a quoted string, e.g. "DATA" or a string expression. The syntax is different to **DOPEN#**, since the **filename** for **OPEN** includes all file attributes, for example: "0:DATA,S,W".

Remarks: For IEC disk units the usage of **DOPEN#** is recommended.

If the first character of the filename is an at sign '@', it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS.

Example: Using **OPEN**

```
OPEN 4,4   :REM OPEN PRINTER
CMD 4      :REM REDIRECT STANDARD OUTPUT TO 4
LIST       :REM PRINT LISTING ON PRINTER DEVICE 4
OPEN 3,8,3,"0:USER FILE,U"
OPEN 2,9,2,"0:DATA,S,W"
```

OR

Format: operand **OR** operand

Usage: Performs a bit-wise logical OR operation on two 16-bit values. Integer operands are used as they are. Real operands are converted to a signed 16-bit integer (losing precision). Logical operands are converted to a 16-bit integer using $FFFF (decimal -1) for TRUE, and $0000 (decimal 0), for FALSE.

Expression	Result
0 OR 0	0
0 OR 1	1
1 OR 0	1
1 OR 1	1

Remarks: The result is of type integer. If the result is used in a logical context, the value of 0 is regarded as FALSE, and all other non-zero values are regarded as TRUE.

Example: Using **OR**

```
PRINT 1 OR 3
3
PRINT 128 OR 64
192
```

In most cases, **OR** is used in **IF** statements.

```
IF (C < 0 OR C > 255) THEN PRINT "NOT A BYTE VALUE"
```

PAINT

Format: **PAINT** x, y, mode [, colour]

Usage: Performs a flood fill of an enclosed graphics area.

x, y is a coordinate pair, which must lie inside the area to be filled.

mode specifies the fill mode:

- **0** use the **colour** to fill the area.
- **1** use the colour of pixel (x,y) to fill the area.

Example: Using **PAINT**

```
 10 GRAPHIC CLR          :REM INITIALISE
 20 SCREEN DEF 1,0,0,2   :REM 320 X 200
 30 SCREEN OPEN 1        :REM OPEN
 40 SCREEN SET 1,1       :REM MAKE SCREEN ACTIVE
 50 PALETTE 1,1,10,15,10 :REM COLOUR 1 TO LIGHT GREEN
 60 PEN 1                :REM SET DRAWING PEN (PEN 0) TO LIGHT GREEN (1)
 70 LINE 160,0,240,100   :REM 1ST. LINE
 80 LINE 240,100,80,100  :REM 2ND. LINE
 90 LINE 80,100,160,0    :REM 3RD. LINE
100 PAINT 160,10,0,1     :REM FILL TRIANGLE WITH COLOUR 1
110 GETKEY K$            :REM WAIT FOR KEY
120 SCREEN CLOSE 1       :REM END GRAPHICS
```

PALETTE

Format: **PALETTE** screen, colour, red, green, blue
PALETTE COLOR colour, red, green, blue
PALETTE RESTORE

Usage: **PALETTE** can be used to change an entry of the system colour palette, or the palette of a screen.
PALETTE RESTORE resets the system palette to the default values.

screen screen number (0-3).

COLOR keyword for changing system palette.

colour index to palette (0-255).

red red intensity (0-15).

green green intensity (0-15).

blue blue intensity (0-15).

Example: Using **PALETTE**

```
10 REM CHANGE SYSTEM COLOUR INDEX
20 REM --- INDEX  9 (BROWN) TO (DARK BLUE)
30 PALETTE COLOR 9,0,0,7
```

```
 10 GRAPHIC CLR            :REM INITIALISE
 20 SCREEN DEF 1,0,0,2     :REM 320 X 200
 30 SCREEN OPEN 1          :REM OPEN
 40 SCREEN SET 1,1         :REM MAKE SCREEN ACTIVE
 50 PALETTE 1,0,  0, 0, 0  :REM 0 = BLACK
 60 PALETTE 1,1, 15, 0, 0  :REM 1 = RED
 70 PALETTE 1,2,  0, 0,15  :REM 2 = BLUE
 80 PALETTE 1,3,  0,15, 0  :REM 3 = GREEN
 90 PEN 2                  :REM SET DRAWING PEN (PEN 0) TO BLUE (2)
100 LINE 160,0,240,100     :REM 1ST. LINE
110 LINE 240,100,80,100    :REM 2ND. LINE
120 LINE 80,100,160,0      :REM 3RD. LINE
130 PAINT 160,10,0,2       :REM FILL TRIANGLE WITH BLUE (2)
140 GETKEY K$              :REM WAIT FOR KEY
150 SCREEN CLOSE 1         :REM END GRAPHICS
```

PEEK

Format: **PEEK**(address)

Usage: Returns an unsigned 8-bit value (byte) from **address**.

If the address is in the range of $0000 to $FFFF (0-65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to $10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

Remarks: Banks 0-127 give access to RAM or ROM banks. Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **PEEK**

```
10 BANK 128              :REM SELECT SYSTEM BANK
20 L = PEEK($02F8)       :REM USR JUMP TARGET LOW
30 H = PEEK($02F9)       :REM USR JUMP TARGET HIGH
40 T = L + 256 * H       :REM 16-BIT JUMP ADDRESS
50 PRINT "USR FUNCTION CALLS ADDRESS";T
```

PEEKW

Format: **PEEKW**(address)

Usage: Returns an unsigned 16-bit value (word) read from **address** (low byte) and **address**+1 (high byte).

If the address is in the range of $0000 to $FFFF (0-65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to $10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

Remarks: Banks 0-127 give access to RAM or ROM banks. Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **PEEKW**

```
20 UA = PEEKW($02F8)   :REM USR JUMP TARGET
50 PRINT "USR FUNCTION CALL ADDRESS";UA
```

PEN

Format: **PEN** [pen,] colour

Usage: Sets the colour of the graphic pen.

pen pen number (0-2):

- **0** drawing pen (default, if only single parameter provided).
- **1** off bits in jam2 mode.
- **2** currently unused.

colour palette index.

Remarks: The colour selected by **PEN** will be used by all graphic/drawing commands that follow it. If you intend to set the drawing **pen** 0 to a

colour, you can omit the first parameter, and only provide the **colour** parameter.

Example: Using **PEN**

```
 10 GRAPHIC CLR           :REM INITIALISE
 20 SCREEN DEF 1,0,0,2    :REM 320 X 200
 30 SCREEN OPEN 1         :REM OPEN
 40 SCREEN SET 1,1        :REM MAKE SCREEN ACTIVE
 50 PALETTE 1,0,  0, 0, 0 :REM 0 = BLACK
 60 PALETTE 1,1, 15, 0, 0 :REM 1 = RED
 70 PALETTE 1,2,  0, 0,15 :REM 2 = BLUE
 80 PALETTE 1,3,  0,15, 0 :REM 3 = GREEN
 90 PEN 1                 :REM SET DRAWING PEN (PEN 0) TO RED (1)
100 LINE 160,0,240,100    :REM DRAW RED LINE
110 PEN 2                 :REM SET DRAWING PEN (PEN 0) TO BLUE (2)
120 LINE 240,100,80,100   :REM DRAW BLUE LINE
130 PEN 3                 :REM SET DRAWING PEN (PEN 0) TO GREEN (3)
140 LINE 80,100,160,0     :REM DRAW GREEN LINE
150 GETKEY K$             :REM WAIT FOR KEY
160 SCREEN CLOSE 1        :REM END GRAPHICS
```

PIXEL

Format: **PIXEL**(x, y)

Usage: Returns the colour of a pixel at the given position.

x absolute screen coordinate.

y absolute screen coordinate.

PLAY

Format: **PLAY** [{string1, string2, string3, string4, string5, string6}]

Usage: **PLAY** without any arguments will cause all voices to be silenced, and all of BASIC's music-system variables to be reset (E.g. **TEMPO**).

PLAY can be followed by up to six comma-separated string arguments, where each argument provides the sequence of notes and directives to be played on a specific voice on the two available SID chips, allowing for up to 6-channel polyphony.

A musical note is a character (A, B, C, D, E, F, or G), which may be preceded by an optional modifier.

Possible modifiers are:

Character	Effect
#	Sharp
$	Flat
.	Dotted
H	Half Note
I	Eighth Note
Q	Quarter Note
R	Pause (rest)
S	Sixteenth Note
W	Whole Note

Embedded directives consist of a letter, followed by a digit:

Char	Directive	Argument Range
O	Octave	0 - 6
T	Instrument Envelope	0 - 9
U	Volume	0 - 9
X	Filter	0 - 1
M	Modulation	0 - 9
P	Portamento	0 - 9
L	Loop	N/A

The modulation directive will modulate your note by the magnitude you specify (1-9), or use 0 to not use it.

Similarly, the portamento directive will gently slide between consecutive notes at the speed you specify (1-9), or use 0 to not use it. Note that the gate-off behaviour of notes is disabled while portamento is enabled, and to re-enable it, you must disable portamento (P0).

Add an **L** directive (no argument needed) at the end of your string if you would like it to loop back to the beginning of your string upon completion.

You have a lot of flexibility on which voice channels you choose to play your melodies on. For instance, you may decide to use only voice 1 and voice 4 for your melody, and spare the other channels for sound effects generated by **SOUND**. Just skip the voices you're not using with **PLAY**, by leaving those arguments empty:

```
PLAY "O4EDCDEEERL",,,"O2CGEGCGEGL"
```

You can even call **PLAY** again to use the aforementioned unused channels, to play another melody alongside your first melody. For example, using voice 2 and voice 5 this time:

```
PLAY ,"O5T2IGAGFEDCEGO6.QCL",,,"O3T2.QG.B O4ICO3GE.QCL"
```

If you wish to assess whether a melody is playing on a voice channel, you can find out by checking the value returned from **RPLAY(voice)**, where the voice parameter is a value from 1 to 6, indicating the voice channel. It will return either 1 (playing), or 0 (not playing).

One caveat to be aware of is that BASIC strings have a maximum length of 255 bytes. If your melody needs to exceed this length, consider breaking up your melody into several strings, then use **RPLAY(voice)** to assess when your first string has finished and then play the next string.

Instrument envelope slots may be modified by using the **ENVELOPE** statement. The default settings for the envelopes are on page 119.

Remarks: The **PLAY** statement makes use of an interrupt driven routine that starts parsing the string and playing the melody. Program execution continues with the next statement, and will not block until the melody has finished.

Example: Using **PLAY**

```
 5 REM *** SIMPLE LOOPING EXAMPLE ***
10 ENVELOPE 9,10,5,10,5,0,300
20 VOL 8
30 TEMPO 30
40 PLAY "O5T9HCIDCDEHCG IGAGFEFDEWCL", "O2T0QCGEGCGEG DBGB CGEGL"
```

```
 5 REM *** MODULATION + PORTAMENTO EXAMPLE ***
10 TEMPO 20
20 M$ = "M5 T2O5P0QD P5FP0RP5QG .AI#AQA HGQE.C IDQE HFQD .DI#CQD HEQ#CQO4HA"
30 M$ = M$ + "O5QDHFQG.AI#AQA HGQE.C IDQEFED#CO4BO5#C DO4AFD P0R L"
40 B$ = "T0QRO2H.D.F.CO1.A.#A.G.A QAIO2AGFE H.D.F.CO1.A.#A.AO2 .D DL"
50 PLAY M$,B$
```

POINTER

Format: **POINTER**(variable)

Usage: Returns the current address of a variable or an array element as a 32-bit pointer. For string variables, it is the address of the string descriptor, not the string itself. The string descriptor consists of three bytes (length, string address low, string address high).

Remarks: The address values of arrays and their elements are constant while the program is executing.
However, the addresses of strings (not their descriptors) may change at any time due to "garbage collection".

Example: Using **POINTER**

```
10 BANK 0                :REM SCALARS ARE IN BANK 0
20 H$="HELLO"            :REM ASSIGN STRING TO H$
30 P=POINTER(H$)         :REM GET DESCRIPTOR ADDRESS
40 PRINT "DESCRIPTOR AT: $";HEX$(P)
50 L=PEEK(P):SP=PEEKW(P+1) :REM LENGTH & STRING POINTER
60 PRINT "LENGTH = ";L   :REM PRINT LENGTH
70 BANK 1                :REM STRINGS ARE IN BANK 1
80 FOR I%=0 TOL-1:PRINT PEEK(SP+I%);:NEXT:PRINT
90 FOR I%=0 TOL-1:PRINT CHR$(PEEK(SP+I%));:NEXT:PRINT

RUN
DESCRIPTOR AT: $F743
LENGTH =  5
 72  69  76  76  79
HELLO
```

POKE

Format: **POKE** address**,** byte [**,** byte ...]

Usage: Writes one or more bytes into memory or memory mapped I/O, starting at **address**.

If the address is in the range of $0000 to $FFFF (0-65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to $10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

byte a value in the range of 0-255.

Remarks: The address is incremented for each data byte, so a memory range can be written to with a single **POKE**.

Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **POKE**

```
10 BANK 128       :REM SELECT SYSTEM BANK
20 POKE $02F8,0,24 :REM SET USR VECTOR TO $1800
```

POKEW

Format: **POKEW** address**,** word [**,** word ...]

Usage: Writes one or more words into memory or memory mapped I/O, starting at **address**.

If the address is in the range of $0000 to $FFFF (0-65535), the memory bank set by **BANK** is used.

Addresses greater than or equal to $10000 (decimal 65536) are assumed to be flat memory addresses and used as such, ignoring the **BANK** setting.

word a value from 0-65535. The first word is stored at address (low byte) and address+1 (high byte). The second word is stored at address+2 (low byte) and address+3 (high byte), etc.

Remarks: The address is increased by two for each data word, so a memory range can be written to with a single **POKEW**.

Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

Example: Using **POKEW**

```
10 BANK 128              :REM SELECT SYSTEM BANK
20 POKEW $02F8,$1800     :REM SET USR VECTOR TO $1800
```

POLYGON

Format: **POLYGON** x**,** y**,** xrad**,** yrad**,** sides [{**,** drawsides**,** subtend**,** angle**,** solid}]

Usage: Draws a regular n-sided polygon. The polygon is drawn using the current drawing context set with **SCREEN**, **PALETTE**, and **PEN**.

x,y centre coordinates.

xrad,yrad radius in x- and y-direction.

sides number of polygon sides.

drawsides sides to draw.

subtend draw line from centre to start (1).

angle start angle.

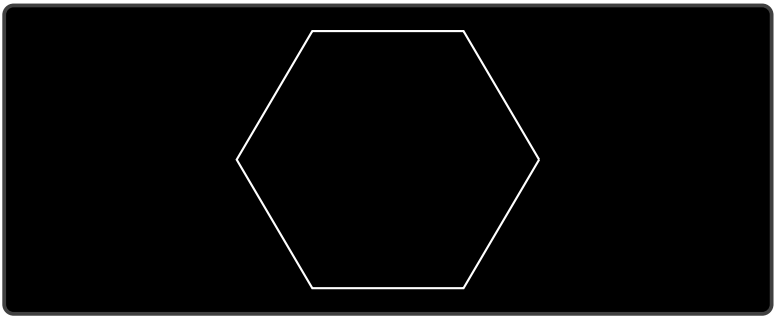
solid fill (1) or outline (0).

Remarks: A regular polygon is both isogonal and isotoxal, meaning all sides and angles are alike.

Example: Using **POLYGON**

```
100 SCREEN 320,200,1        :REM OPEN 320 x 200 SCREEN
110 POLYGON 160,100,40,40,6 :REM DRAW HONEYCOMB
120 GETKEY A$               :REM WAIT FOR KEY
130 SCREEN CLOSE            :REM CLOSE GRAPHICS SCREEN
```

Results in:



POS

Format: **POS**(dummy)

Usage: Returns the cursor column relative to the currently used window.

dummy a numeric value, which is ignored.

Remarks: **POS** gives the column position for the screen cursor. It will not work for redirected output.

Example: Using **POS**

```
10 IF POS(0) > 72 THEN PRINT :REM INSERT RETURN
```

POT

Format: **POT**(paddle)

Usage: Returns the position of a paddle.

paddle paddle number (1-4).

The low byte of the return value is the paddle value, with 0 at the clockwise limit and 255 at the anticlockwise limit.

A value greater than 255 indicates that the fire button is also being pressed.

Remarks: Analogue paddles are noisy and inexact. The range may be less than 0-255 and there could be some jitter in the values returned from **POT**.

Example: Using **POT**

```
10 X = POT(1)       : REM READ PADDLE #1
20 B = X > 255      : REM TRUE (-1) IF FIRE BUTTON IS PRESSED
30 V = X AND 255    : REM PADDLE #1 VALUE
```

PRINT

Format: **PRINT** arguments

Usage: Evaluates the argument list, and prints the values formatted to the current screen window. Standard formatting is used, depending on the argument type. For user controlled formatting, see **PRINT USING**.

The following argument types are evaluated:

- **numeric** the printout starts with a space for positive and zero values, or a minus sign for negative values. Integer values are printed with the necessary number of digits. Real values are printed in either fixed point form (typically 9 digits), or scientific form if the value is outside the range of 0.01 to 999999999.
- **string** the string may consist of printable characters and control codes. Printable characters are printed at the cursor position, while control codes are executed.
- , a comma acts as a tabulator.
- ; a semicolon acts as a separator between arguments of the list. Other than the comma character, it does not insert any additional characters. A semicolon at the end of the argument list suppresses the automatic return (carriage return) character.

Remarks: The **SPC** and **TAB** functions may be used in the argument list for positioning. **CMD** can be used for redirection.

Example: Using **PRINT**

```
10 FOR I=1 TO 10    : REM START LOOP
20 PRINT I,I*I,SQR(I)
30 NEXT
```

PRINT#

Format: **PRINT#** channel**,** arguments

Usage: Evaluates the argument list, and prints the formatted values to the device assigned to **channel**. Standard formatting is used, depending on the argument type. For user controlled formatting, see **PRINT# USING**.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**.

The following argument types are evaluated:

- **numeric** the printout starts with a space for positive and zero values, or a minus sign for negative values. Integer values are printed with the necessary number of digits. Real values are printed in either fixed point form (typically 9 digits), or scientific form if the value is outside the range of 0.01 to 999999999.
- **string** may consist of printable characters and control codes. Printable characters are printed at the cursor position, while control codes are executed.
- , a comma acts as a tabulator.
- ; a semicolon acts as a separator between arguments of the list. Other than the comma character, it does not insert any additional characters. A semicolon at the end of the argument list suppresses the automatic return (carriage return) character.

Remarks: The **SPC** and **TAB** functions are not suitable for devices other than the screen.

Example: Using **PRINT#** to write a file to drive 8:

```
10 DOPEN#2,"TABLE",W,U8
20 FOR I=1 TO 10    : REM START LOOP
30 PRINT#2,I,I*I,SQR(I)
40 NEXT
50 DCLOSE#2
```

You can confirm that the file **'TABLE'** has been written by typing `DIR "TA*"`, and then view the contents of the file by typing `TYPE "TABLE"`.

PRINT USING

Format: **PRINT**[**#** channel**,**] **USING** format**;** argument

Usage: Parses the **format** string and evaluates the argument. The argument can be either a string or a numeric value. The format of the resulting output is directed by the **format** string.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**. If no channel is specified, the output goes to the screen.

format string variable or a string constant which defines the rules for formatting. When using a number as the **argument**, formatting can be done in either CBM style, providing a pattern such as ####.## or in C style using a <width.precision> specifier, such as %3D %7.2F %4X .

argument the number to be formatted. If the argument does not fit into the format e.g. trying to print a 4 digit variable into a series of three hashes (###), asterisks will be used instead.

Remarks: The format string is applied for one argument only, but it is possible to append more with **USING format;argument** sequences.

argument may consist of printable characters and control codes. Printable characters are printed to the cursor position, while control codes are executed. The number of # characters sets the width of the output. If the first character of the format string is an equals '=' sign, the argument string is centered. If the first character of the format string is a greater than '>' sign, the argument string is right justified.

Examples: Using **PRINT# USING**

```
PRINT USING "##.##";π, USING " [%6.4F] ";SQR(2)
 3.14 [1.4142]

PRINT USING " ( # # # ) ";12*31
 ( 3 7 2 )

PRINT USING "###"; "ABCDE"
ABC

PRINT USING ">###"; "ABCDE"
CDE

PRINT USING "ADDRESS:$%4X";65000
ADDRESS:$FDE8

A$="####,###,###.#":PRINT USING A$;1E8/3
 33,333,333.3
```

RCOLOR

Format: **RCOLOR**(colour source)

Usage: Returns the current colour index for the selected colour source.

Colour sources are:

- **0** background colour (VIC $D021).
- **1** text colour ($F1).
- **2** highlight colour ($2D8).
- **3** border colour (VIC $D020).

Example: Using **RCOLOR**

```
10 C = RCOLOR(3)  : REM C = colour index of border colour
```

RCURSOR

Format: **RCURSOR** {colvar**,** rowvar}

Usage: Returns the current cursor column and row.

Remarks: The row and column values start at zero, where the left-most column is zero, and the top row is zero.

Example: Using **RCURSOR**

```
100 CURSOR ON,20,10
110 PRINT "[HERE]";
120 RCURSOR X,Y
130 PRINT " COL:";X;" ROW:";Y

RUN

                [HERE] COL: 26  ROW: 10
```

READ

Format: **READ** variable [**,** variable ...]

Usage: Reads values from program source into variables.

variable list Any legal variables.

All types of constants (integer, real, and strings) can be read, but not expressions. Items are separated by commas. Strings containing commas, colons or spaces must be put in quotes.

RUN initialises the data pointer to the first item of the first **DATA** statement and advances it for every read item. It is the programmer's responsibility that the type of the constant and the variable in the **READ** statement match. Empty items with no constant between commas are allowed and will be interpreted as zero for numeric variables and an empty string for string variables.

RESTORE may be used to set the data pointer to a specific line for subsequent readings.

Remarks: It is good programming practice to put large amounts of **DATA** statements at the end of the program, so they don't slow down the search for line numbers after **GOTO**, and other statements with line number targets.

Example: Using **READ**

```
10 READ NA$, VE
20 READ N%:FOR I=2 TO N%:READ GL(I):NEXT I
30 PRINT "PROGRAM:";NA$;"   VERSION:";VE
40 PRINT "N-POINT GAUSS-LEGENDRE FACTORS E1":
50 FOR I=2 TO N%:PRINT I;GL(I):NEXT I
30 STOP
80 DATA "MEGA65",1.1
90 DATA 5,0.5120,0.3573,0.2760,0.2252
```

RECORD

Format: **RECORD#** channel**,** record [**,** byte]

Usage: Positions the read/write pointer of a relative file.

channel number, which was given to a previous call of commands such as **DOPEN**, or **OPEN**.

record target record (1-65535).

byte byte position in record.

RECORD can only be used for files of type **REL**, which are relative files capable of direct access.

RECORD positions the file pointer to the specified record number. If this record number does not exist and there is enough space on the disk which **RECORD** is writing to, the file is expanded to the requested

record count by adding empty records. When this occurs, the disk status will give the message RECORD NOT PRESENT, but this is not an error!

After a call of **INPUT#** or **PRINT#**, the file pointer will proceed to the next record position.

Remarks: The Commodore disk drives have a bug in their DOS, which can destroy data by using relative files. A recommended workaround is to use the command **RECORD** twice, before and after the I/O operation.

Example: Using **RECORD**

```
100 DOPEN#2,"DATA BASE",L240 :REM OPEN OR CREATE
110 FOR I%=1 TO 20           :REM WRITE LOOP
120 PRINT#2,"RECORD #";I%    :REM WRITE RECORD
130 NEXT I%                  :REM END LOOP
140 DCLOSE#2                 :REM CLOSE FILE
150                          :REM NOW TESTING
160 DOPEN#2,"DATA BASE",L240 :REM REOPEN
170 FOR I%=20 TO 2 STEP -2   :REM READ FILE BACKWARDS
180 RECORD#2,I%              :REM POSITION TO RECORD
190 INPUT#2,A$               :REM READ RECORD
200 PRINT A$;:IF I% AND 2 THEN PRINT
210 NEXT I%                  :REM LOOP
220 DCLOSE#2                 :REM CLOSE FILE

RUN
RECORD # 20 RECORD # 18
RECORD # 16 RECORD # 14
RECORD # 12 RECORD # 10
RECORD # 8 RECORD # 6
RECORD # 4 RECORD # 2
```

REM

Format: **REM**

Usage: Marks any characters after **REM** on the same line as a comment.

Characters after **REM** are never executed, they're ignored by BASIC.

Example: Using **REM**

```
10 REM *** PROGRAM TITLE ***
20 N=1000 :REM NUMBER OF ITEMS
30 DIM NA$(N)
```

RENAME

Format: **RENAME** old **TO** new [**,D** drive] [**,U** unit]

Usage: Renames a disk file.

old is either a quoted string, e.g. "DATA" or a string expression in brackets, e.g. (FI$).

new is either a quoted string, e.g. "BACKUP" or a string expression in brackets, e.g. (FS$)

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **RENAME** is executed in the DOS of the disk drive. It can rename all regular file types (**PRG**, **REL**), **SEQ**, **USR**. The old file must exist, and the new file must not exist. Only single files can be renamed, wildcard characters such as '*' and '?' are not allowed. The file type cannot be changed.

Example: Using **RENAME**

```
RENAME "CODES" TO "BACKUP" :REM RENAME SINGLE FILE
```

RENUMBER

Format: **RENUMBER** [{new**,** inc**,** range}]

Usage: Used to renumber all, or a range of lines of a BASIC program.

new new starting line of the line range to renumber. The default value is 10.

inc increment to be used. The default value is 10.

range line range to renumber. The default values are from first to last line.

RENUMBER changes all line numbers in the chosen range and also changes all references in statements that use **GOSUB**, **GOTO**, **RESTORE**, **RUN**, **TRAP**, etc.

RENUMBER can only be executed in direct mode. If it detects a problem such as memory overflow, unresolved references or line number overflow (more than than 64000 lines), it will stop with an error message and leave the program unchanged.

RENUMBER may be called with 0-3 parameters. Unspecified parameters use their default values.

Remarks: **RENUMBER** may need several minutes to execute for large programs.

Examples: Using **RENUMBER**

```
RENUMBER               :REM NUMBERS WILL BE 10,20,30,...
RENUMBER 100,5         :REM NUMBERS WILL BE 100,105,110,115,...
RENUMBER 601,1,500     :REM RENUMBER STARTING AT 500 TO 601,602,...
RENUMBER 100,5,120-180 :REM RENUMBER LINES 120-180 TO 100,105,...
```

RESTORE

Format: **RESTORE** [line]

Usage: Set, or reset the internal pointer for **READ** from **DATA** statements.

line new position for the pointer. The default is the first program line.

Remarks: The new pointer target **line** does not need to contain **DATA** statements. Every **READ** will advance the pointer to the next **DATA** statement automatically.

Example: Using **RESTORE**

```
10 DATA 3,1,4,1,5,9,2,6
20 DATA "MEGA65"
30 DATA 2,7,1,8,2,8,9,5
40 FOR I=1 TO 8:READ P:PRINT P:NEXT
50 RESTORE 30
60 FOR I=1 TO 8:READ P:PRINT P:NEXT
70 RESTORE 20
80 READ A$:PRINT A$
```

RESUME

Format: **RESUME** [line | **NEXT**]

Usage: Used in a **TRAP** routine to resume normal program execution after handling an error.

RESUME with no parameters attempts to re-execute the statement that caused the error. The **TRAP** routine should have examined and corrected the issue where the error occurred.

line line number to resume program execution at.

NEXT resumes execution following the statement that caused the error. This could be the next statement on the same line (separated with a colon ':'), or the statement on the next line.

Remarks: **RESUME** cannot be used in direct mode.

Example: Using **RESUME**

```
 10 TRAP 100
 20 FOR I=1 TO 100
 30 PRINT EXP(I)
 40 NEXT
 50 PRINT "STOPPED FOR I =";I
 60 END
100 PRINT ERR$(ER): RESUME 50
```

RETURN

Format: **RETURN**

Usage: Returns control from a subroutine, which was called with **GOSUB** or an event handler declared with **COLLISION**.

The execution continues at the statement following the **GOSUB** call.

In the case of the **COLLISION** handler, the execution continues at the statement where it left from to call the handler.

Example: Using **RETURN**

```
 10 SCNCLR          :REM CLEAR SCREEN
 20 FOR I=1 TO 20   :REM DEFINE LOOP
 30 GOSUB 100       :REM CALL SUBROUTINE
 40 NEXT I          :REM LOOP
 50 END             :REM END OF PROGRAM
100 CURSOR ON,I,I,0 :REM ACTIVATE AND POSITION CURSOR
110 PRINT "X";      :REM PRINT X
120 SLEEP 0.5       :REM WAIT 0.5 SECONDS
130 CURSOR OFF      :REM SWITCH BLINKING CURSOR OFF
140 RETURN          :REM RETURN TO CALLER
```

RGRAPHIC

Format: **RGRAPHIC**(screen**,** parameter**)**

Usage: Return graphic screen status and parameters

Parameter	Description
0	Open (1), Closed (0), or Invalid (>1)
1	Width (0=320, 1=640)
2	Height (0=200, 1=400)
3	Depth (1-8 Bitplanes)
4	Bitplanes Used (Bitmask)
5	Bank 4 Blocks Used (Bitmask)
6	Bank 5 Blocks Used (Bitmask)
7	Drawscreen # (0-3)
8	Viewscreen # (0-3)
9	Drawmodes (Bitmask)
10	pattern type (bitmask)

Example: Using **RGRAPHIC**

```
 10 GRAPHIC CLR         :REM INITIALISE
 20 SCREEN DEF 0,1,0,4  :REM SCREEN 0:640 X 200 X 4
 30 SCREEN OPEN 0       :REM OPEN
 40 SCREEN SET 0,0      :REM DRAW = VIEW = 0
 50 SCNCLR 0            :REM CLEAR
 60 PEN 0,1             :REM SELECT COLOUR
 70 LINE 0,0,639,199    :REM DRAW LINE
 80 FOR I=0 TO 10:A(I)=RGRAPHIC(0,I) :NEXT
 90 SCREEN CLOSE 0
100 FOR I=0 TO  6:PRINT I;A(I):NEXT :REM PRINT INFO

RUN
 0  1
 1  1
 2  0
 3  4
 4  15
 5  15
 6  15
```

RIGHT$

Format: **RIGHT$**(string, n)

Usage: Returns a string containing the last **n** characters from **string**. If the length of **string** is equal or less than **n**, the result string will be identical to the argument string.

string a string expression.

n a numeric expression (0-255).

Remarks: Empty strings and zero lengths are legal values.

Example: Using **RIGHT$**:

```
PRINT RIGHT$("MEGA-65",2)
65
```

RMOUSE

Format: **RMOUSE** x variable, y variable, button variable

Usage: Reads mouse position and button status.

x variable numeric variable where the x-position will be stored.

y variable numeric variable where the y-position will be stored.

button variable numeric variable receiving button status.
left button sets bit 7, while right button sets bit 0.

Value	Status
0	No Button
1	Right Button
128	Left Button
129	Both Buttons

RMOUSE places -1 into all variables if the mouse is not connected or disabled.

Remarks: Active mice on both ports merge the results.

Example: Using **RMOUSE**:

```
10 MOUSE ON, 1, 1        :REM MOUSE ON PORT 1 WITH SPRITE 1
20 RMOUSE XP, YP, BU     :REM READ MOUSE STATUS
30 IF XP < 0 THEN PRINT "NO MOUSE ON PORT 1":STOP
40 PRINT "MOUSE:";XP;YP;BU
50 MOUSE OFF             :REM DISABLE MOUSE
```

RND

Format: **RND**(type)

Usage: Returns a pseudo random number.

This is called a "pseudo" random number, as the numbers are not *really* random. They are derived from another number called a "seed" that generates reproducible sequences. **type** determines which seed is used:

- **type = 0** use system clock.
- **type < 0** use the value of **type** as seed.
- **type > 0** derive a new random number from previous one.

Remarks: Seeded random number sequences produce the same sequence for identical seeds.

Example: Using **RND**:

```
10 DEF FNDI(X) = INT(RND(0)*6)+1 :REM DICE FUNCTION
20 FOR I=1 TO 10                 :REM THROW 10 TIMES
30 PRINT I;FNDI(0)               :REM PRINT DICE POINTS
40 NEXT
```

RPALETTE

Format: **RPALETTE**(screen**,** index**,** rgb)

Usage: Returns the red, green or blue value of a palette colour index.

screen screen number (0-3).

index palette colour index.

rgb (0: red, 1: green, 2:blue).

Example: Using **RPALETTE**

```
10 SCREEN 320,200,4    :REM DEFINE AND OPEN SCREEN
20 R = RPALETTE(0,3,0) :REM GET RED
30 G = RPALETTE(0,3,1) :REM GET GREEN
40 B = RPALETTE(0,3,2) :REM GET BLUE
50 SCREEN CLOSE        :REM CLOSE SCREEN
60 PRINT "PALETTE INDEX 3 RGB =";R;G;B

RUN
PALETTE INDEX 3 RGB = 0  15  15
```

RPEN

Format: **RPEN**(n)

Usage: Returns the colour index of pen n.

n pen number (0-2), where:

- 0 draw pen.
- 1 erase pen.
- 2 outline pen.

Example: Using **RPEN**

```
 10 GRAPHIC CLR          :REM INITIALISE
 20 SCREEN DEF 0,1,0,4   :REM SCREEN 0:640 X 200 X 4
 30 SCREEN OPEN 0        :REM OPEN
 40 SCREEN SET 0,0       :REM DRAW = VIEW = 0
 50 SCNCLR 0             :REM CLEAR
 60 PEN 0,1              :REM SELECT COLOUR
 70 X = RPEN(0)
 80 Y = RPEN(1)
 90 C = RPEN(2)
100 SCREEN CLOSE 0
110 PRINT "DRAW PEN COLOUR = ";X
RUN
DRAW PEN COLOUR =  1
```

RPLAY

Format: **RPLAY**(voice)

Usage: Returns a value of 1 or 0, to indicate whether a melody is playing on the given voice channel or not.

voice the voice channel to assess, ranging from 1 to 6.

Example: Using **RPLAY**:

```
 10 PLAY "O4ICDEFGAB05CR","O2QCGEGC01GCR"
 30 IF RPLAY(1) OR RPLAY(2) THEN GOTO 30: REM WAIT FOR END OF SONG
```

RREG

Format: **RREG** [{areg, xreg, yreg, zreg, sreg}]

Usage: Reads the values that were in the CPU registers after a **SYS** call, into the specified variables.

areg gets accumulator value.

xreg gets X register value.

yreg gets Y register value.

zreg gets Z register value.

sreg gets status register value.

Remarks: The register values after a **SYS** call are stored in system memory. This is how **RREG** is able to retrieve them.

Example: Using **RREG**:

```
10 BANK 128
20 BLOAD "ML PROG",8192
30 SYS 8192
40 RREG A,X,Y,Z,S
50 PRINT "REGISTER:";A;X;Y;Z;S
```

RSPCOLOR

Format: **RSPCOLOR**(n)

Usage: Returns multi-colour sprite colours.

n sprite multi-colour number:

- **1** get multi-colour # 1.
- **2** get multi-colour # 2.

Remarks: Refer to **SPRITE** and **SPRCOLOR** for more information.

Example: Using **RSPCOLOR**:

```
10 SPRITE 1,1         :REM TURN SPRITE 1 ON
20 C1% = RSPCOLOR(1)  :REM READ COLOUR #1
30 C2% = RSPCOLOR(2)  :REM READ COLOUR #2
```

RSPEED

Format: **RSPEED**(n)

Usage: Returns the current CPU clock in MHz.

n numeric dummy argument, which is ignored.

Remarks: **RSPEED(n)** will not return the correct value if `POKE 0,65` has previously been used to enable the highest speed (40MHz).

Refer to the **SPEED** command for more information.

Example: Using **RSPEED**:

```
10 X=RSPEED(0)      :REM GET CLOCK
20 IF X=1  THEN PRINT "1 MHZ"   :GOTO 50
30 IF X=3  THEN PRINT "3.5 MHZ" :GOTO 50
40 IF X=40 THEN PRINT "40 MHZ"
50 END
```

RSPPOS

Format: **RSPPOS**(sprite, n)

Usage: Returns a sprite's position and speed

sprite sprite number.

n sprite parameter to retrieve:

- **0** X position.
- **1** Y position.
- **2** speed.

Remarks: Refer to the **MOVSPR** and **SPRITE** commands for more information.

Example: Using **RSPPOS**:

```
10 SPRITE 1,1        :REM TURN SPRITE 1 ON
20 XP = RSPPOS(1,0)  :REM GET X OF SPRITE 1
30 YP = RSPPOS(1,1)  :REM GET Y OF SPRITE 1
30 SP = RSPPOS(1,2)  :REM GET SPEED OF SPRITE 1
```

RSPRITE

Format: **RSPRITE**(sprite, n)

Usage: Returns a sprite's parameter.

sprite sprite number (0-7).

n the sprite parameter to return (0-5):

- **0** turned on (0 or 1) A 0 means the sprite is off.
- **1** foreground colour (0-15).
- **2** background priority (0 or 1).
- **3** x-expanded (0 or 1). 0 means it's not expanded.
- **4** y-expanded (0 or 1). 0 means it's not expanded.

- **5** multi-colour (0 or 1). 0 means it's not multi-colour.

Remarks: Refer to the **MOVSPR** and **SPRITE** commands for more information.

Example: Using **RSPRITE**:

```
10 SPRITE 1,1          :REM TURN SPRITE 1 ON
20 EN = RSPRITE(1,0)   :REM SPRITE 1 ENABLED ?
30 FG = RSPRITE(1,1)   :REM SPRITE 1 FOREGROUND COLOUR INDEX
40 BP = RSPRITE(1,2)   :REM SPRITE 1 BACKGROUND PRIORITY
50 XE = RSPRITE(1,3)   :REM SPRITE 1 X EXPANDED ?
60 YE = RSPRITE(1,4)   :REM SPRITE 1 Y EXPANDED ?
70 MC = RSPRITE(1,5)   :REM SPRITE 1 MULTI-COLOUR ?
```

RUN

Format: **RUN** [line number]
RUN filename [**,D** drive] [**,U** unit]

Usage: Run a BASIC program.

If a filename is given, the program file is loaded into memory and run, otherwise the program that is currently in memory will be used instead.

line number an existing line number of the program in memory to run from.

filename either a quoted string, e.g. `"PROG"` or a string expression in brackets, e.g. `(PR$)`. The filetype must be **PRG**.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

RUN first resets all internal pointers to their default values. Therefore, there will be no variables, arrays or strings defined. The run-time stack is also reset, and the table of open files is cleared.

Remarks: To start or continue program execution without resetting everything, use **GOTO** instead.

Examples: Using **RUN**

```
RUN "FLIGHTSIM" :REM LOAD AND RUN PROGRAM FLIGHTSIM
RUN 1000        :REM RUN PROGRAM IN MEMORY, START AT LINE# 1000
RUN             :REM RUN PROGRAM IN MEMORY
```

RWINDOW

Format: **RWINDOW**(n)

Usage: Returns information regarding the current text window.

n the screen parameter to retrieve:

- **0** width of current text window.
- **1** height of current text window.
- **2** number of columns on screen (40 or 80).

Remarks: Older versions of **RWINDOW** reported the width - 1 and the height - 1 for arguments 0 and 1.

Refer to the **WINDOW** command for more information.

Example: Using **RWINDOW**:

```
10 W = RWINDOW(2)          :REM GET SCREEN WIDTH
20 IF W=80 THEN BEGIN      :REM IS 80 COLUMNS MODE ACTIVE?
30   PRINT CHR$(27)+"X";   :REM YES, SWITCH TO 40COLUMNS
40 BEND
```

SAVE

Format: **SAVE** filename [, unit]
[←] filename [, unit]

Usage: Saves a BASIC program to a file of type **PRG**.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

The maximum length of the filename is 16 characters, not counting the optional save and replace character '@' and the in-file drive definition. If the first character of the filename is an at sign '@', it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS. The filename may be preceded by the drive number definition "0:" or "1:", which is only relevant for dual drive disk units.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **SAVE** is obsolete, implemented only for backwards compatibility. **DSAVE** should be used instead. The shortcut symbol [←] is next to [1]. Can only be used in direct mode.

Examples: Using **SAVE**

```
SAVE "ADVENTURE"
SAVE "ZORK-I",8
SAVE "1:DUNGEON",9
```

SAVEIFF

Format: **SAVEIFF** filename [**,D** drive] [**,U** unit]

Usage: Saves a picture from memory to a disk file in **IFF** format. The IFF (Interchange File Format) is supported by many different applications and operating systems. **SAVEIFF** saves the image, the palette and resolution parameters.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI$). The maximum length of the filename is 16 characters. If the first character of the filename is an at sign '@' it is interpreted as a "save and replace" operation. It is not recommended to use this option on 1541 and 1571 drives, as they contain a "save and replace bug" in their DOS.

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: Files saved with **SAVEIFF** can be loaded with **LOADIFF**. Tools are available to convert popular image formats to IFF. These tools are available on several operating systems, such as AMIGA OS, macOS, Linux, and Windows. For example, **ImageMagick** is a free graphics package that includes a tool called **convert**, which can be used to create IFF files in conjunction with the **ppmtoilbm** tool from the **Netbpm** package.

Example: Using **SAVEIFF**

```
10 SCREEN 320,200,2          :REM SCREEN #0 320 X 200 X 2
20 PEN 1                     :REM DRAWING PEN COLOR 1 (WHITE)
30 LINE 25,25,295,175        :REM DRAW LINE
40 SAVEIFF "LINE-EXAMPLE",U8 :REM SAVE CURRENT VIEW TO FILE
50 SCREEN CLOSE              :REM CLOSE SCREEN AND RESTORE PALETTE
```

SCNCLR

Format: **SCNCLR** [colour]

Usage: Clears a text window or screen.

SCNCLR (with no arguments) clears the current text window. The default window occupies the whole screen.

SCNCLR colour clears the graphic screen by filling it with the given **colour**.

Example: Using **SCNCLR**:

```
1 REM SCREEN EXAMPLE 2
10 GRAPHIC CLR           :REM INITIALIZE
20 SCREEN DEF 1,0,0,2    :REM SCREEN #1 320 X 200 X 2
30 SCREEN OPEN 1         :REM OPEN SCREEN 1
40 SCREEN SET 1,1        :REM USE SCREEN 1 FOR RENDERING AND VIEWING
50 SCREEN CLR 0          :REM CLEAR SCREEN
60 PALETTE 1,1,15,15,15  :REM DEFINE COLOUR 1 AS WHITE
70 PEN 0,1               :REM DRAWING PEN
80 LINE 25,25,295,175    :REM DRAW LINE
90 SLEEP 10              :REM WAIT FOR 10 SECONDS
100 SCREEN CLOSE 1       :REM CLOSE SCREEN AND RESTORE PALETTE
```

SCRATCH

Format: **SCRATCH** filename [,**D** drive] [,**U** unit] [,**R**]

Usage: Used to erase a disk file.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (F1$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

R Recover a previously erased file. This will only work if there were no write operations between erasure and recovery, which may have altered the contents of the file.

Remarks: **SCRATCH filename** is a synonym of **ERASE**.

The success and the number of erased files can be examined by **PRINT**ing or using the system variable **DS$**. The second last number, which normally reports the track number in case of a disk error, instead reports the number of successfully erased files.

Examples: Using **SCRATCH**

```
SCRATCH "DRM",U9 :REM SCRATCH FILE DRM ON UNIT 9
PRINT DS$
01, FILES SCRATCHED,01,00

SCRATCH "OLD*"   :REM SCRATCH ALL FILES BEGINNING WITH "OLD"
PRINT DS$
01, FILES SCRATCHED,04,00
```

SCREEN

Format: **SCREEN** [screen,] width, height, depth
SCREEN CLR colour
SCREEN DEF width flag, height flag, depth
SCREEN SET drawscreen, viewscreen
SCREEN OPEN [screen]
SCREEN CLOSE [screen]

Usage: There are two approaches available when preparing the screen for the drawing of graphics: a simplified approach, and a detailed approach.

Simplified approach:

The first version of **SCREEN** (which has pixel units for width and height) is the easiest way to start a graphics screen, and is the preferred method if only a single screen is needed (i.e., a second screen isn't needed for double buffering). This does all of the preparatory work for you, and will call commands such as **GRAPHIC CLR**, **SCREEN CLR**, **SCREEN DEF**, **SCREEN OPEN** and, **SCREEN SET** on your behalf. It takes the following parameters:

SCREEN [screen,] width, height, depth

- **screen** the screen number (0-3) is optional. If no screen number is given, screen 0 is used. To keep this approach as simple as possible, it is suggested to use the default screen 0.
- **width** 320 or 640 (default 320)
- **height** 200 or 400 (default = 200)
- **depth** 1..8 (default = 8), colours = 2 ^depth.

The argument parser is error tolerant and uses default values for width (320) and height (200) if the parsed argument is not valid.

This version of **SCREEN** starts with a predefined palette and sets the background to black, and the pen to white, so drawing can start immediately using the default values.

On the other hand, the detailed approach will require the setting of palette colours and pen colour before any drawing can be done.

The **colour** value must be in the range of 0 to 15. Refer to the colour table under **BACKGROUND** on page 79 for the **colour** values and their corresponding colours.

When you are finished with your graphics screen, simply call **SCREEN CLOSE** [screen] to return to the text screen.

Detailed approach:

The other versions of **SCREEN** perform special actions, used for advanced graphics programs that open multiple screens, or require "double buffering". If you have chosen the simplified approach, you will not require any of these versions below, apart from **SCREEN CLOSE**.

SCREEN CLR colour (or **SCNCLR** colour)
Clears the active graphics screen by filling it with **colour**.

SCREEN DEF screen, width flag, height flag, depth
Defines resolution parameters for the chosen screen. The width flag and height flag indicate whether high resolution (1) or low resolution (0) is chosen.

- **screen** screen number 0-3
- **width flag** 0-1 (0:320, 1:640 pixel)
- **height flag** 0-1 (0:200, 1:400 pixel)
- **depth** 1-8 (2 - 256 colours)

Note that the width and height values here are **flags**, and **not pixel units**.

SCREEN SET drawscreen, viewscreen
Sets screen numbers (0-3) for the drawing and the viewing screen, i.e., while one screen is being viewed, you can draw on a separate screen and then later flip between them. This is what's known as double buffering.

SCREEN OPEN screen
Allocates resources and initialises the graphics context for the selected **screen** (0-3). An optional variable name as a further argument, gets the result of the command that can be tested afterwards for success.

SCREEN CLOSE [screen]
Closes **screen** (0-3) and frees resources. If no value is given, it will default to 0. Also note that upon closing a screen, **PALETTE RESTORE** is automatically performed for you.

Examples: Using **SCREEN**:

```
 5 REM *** SIMPLIFIED APPROACH ***
10 SCREEN 320,200,2    :REM SCREEN #0: 320 X 200 X 2
20 PEN 1               :REM DRAWING PEN COLOUR = 1 (WHITE)
30 LINE 25,25,295,175  :REM DRAW LINE
40 GETKEY A$           :REM WAIT KEYPRESS
50 SCREEN CLOSE        :REM CLOSE SCREEN 0 (RESTORE PALETTE)
```

```
 5 REM *** DETAILED APPROACH ***
10 GRAPHIC CLR         :REM INITIALISE
20 SCREEN DEF 1,0,0,2  :REM SCREEN #1: 320 X 200 X 2
30 SCREEN OPEN 1       :REM OPEN SCREEN 1
40 SCREEN SET 1,1      :REM USE SCREEN 1 FOR RENDERING AND VIEWING
50 SCREEN CLR 0        :REM CLEAR SCREEN
60 PALETTE 1,1,15,15,15:REM DEFINE COLOUR 1 AS WHITE
70 PEN 0,1             :REM DRAWING PEN
80 LINE 25,25,295,175  :REM DRAW LINE
90 SLEEP 10            :REM WAIT 10 SECONDS
100 SCREEN CLOSE 1     :REM CLOSE SCREEN 1 (RESTORE PALETTE)
```

SET

Format: **SET DEF** unit
SET DISK old **TO** new
SET VERIFY <**ON** | **OFF**>

Usage: **SET DEF** redefines the default unit for disk access, which is initialised to 8 by the DOS. Commands that do not explicitly specify a unit will use this default unit.

SET DISK is used to change the unit number of a disk drive temporarily.

SET VERIFY enables or disables the DOS verify-after-write mode for 3.5 drives.

Remarks: These settings are valid until a reset or shutdown.

Examples: Using **SET**:

```
DIR             :REM SHOW DIRECTORY OF UNIT 8
SET DEF 11      :REM UNIT 11 BECOMES DEFAULT
DIR             :REM SHOW DIRECTORY OF UNIT 11
DLOAD "*"       :REM LOAD FIRST FILE FROM UNIT 11
SET DISK 8 TO 9 :REM CHANGE UNIT# OF DISK DRIVE 8 TO 9
DIR U9          :REM SHOW DIRECTORY OF UNIT 9 (FORMER 8)
SET VERIFY ON   :REM ACTIVATE VERIFY-AFTER-WTITE MODE
```

SGN

Format: **SGN**(numeric expression)

Usage: Extracts the sign from the argument and returns it as a number:

- **-1** negative argument.
- **-0** zero.
- **1** positive, non-zero argument.

Example: Using **SGN**

```
10 ON SGN(X)+2 GOTO 100,200,300 :REM TARGETS FOR MINUS,ZERO,PLUS
20 Z = SGN(X) * ABS(Y) : REM COMBINE SIGN OF X WITH VALUE OF Y
```

SIN

Format: **SIN**(numeric expression)

Usage: Returns the sine of the **numeric expression**. The argument is expected in units of radians. The result is in the range (-1.0 to +1.0)

Remarks: An argument in units of degrees can be converted to radians by multiplying it with $\pi/180$.

Examples: Using **SIN**

```
PRINT SIN(0.7)
 .644217687

X=30:PRINT SIN(X * π / 180)
 .5
```

SLEEP

Format: **SLEEP** seconds

Usage: Pauses execution for the given duration. The argument is a positive floating point number. The precision is 1 microsecond.

Remarks: Pressing RUN STOP interrupts the sleep.

Example: Using **SLEEP**

```
20 SLEEP 10     :REM WAIT 10 SECONDS
40 SLEEP 0.0005 :REM SLEEP 500 MICRO SECONDS
50 SLEEP 0.01   :REM SLEEP  10 MILLI SECONDS
60 SLEEP DD     :REM TAKE SLEEP TIME FROM VARIABLE DD
70 SLEEP 600    :REM SLEEP 10 MINUTES
```

SOUND

Format: **SOUND** voice**,** freq**,** dur [{**,** dir**,** min**,** sweep**,** wave **,** pulse}]

Usage: Plays a sound effect.

voice voice number (1-6).

freq frequency (0-65535).

dur duration in jiffies (0-32767). The duration of a jiffy depends on the display standard. There are 50 jiffies per second with PAL, 60 per second with NTSC.

dir direction (0:up, 1:down, 2:oscillate).

min minimum frequency (0-65535).

sweep sweep range (0-65535).

wave waveform (0:triangle, 1:sawtooth, 2:square, 3:noise).

pulse pulse width (0-5095).

Remarks: **SOUND** starts playing the sound effect and immediately continues with the execution of the next BASIC statement while the sound effect is played. This enables the showing of graphics or text and playing sounds simultaneously.

Examples: Using **SOUND**

```
IF PEEK($D06F) AND $80 THEN J = 60: ELSE J = 50 :REM J IS JIFFIES PER SECOND
SOUND 1, 7382, J                    :REM PLAY SQUARE WAVE ON VOICE 1 FOR 1 SECOND
SOUND 2,  800, J*60                 :REM PLAY SQUARE WAVE ON VOICE 2 FOR 1 MINUTE
SOUND 3, 4000, 120, 2, 2000, 400, 1 :REM PLAY SWEEPING SAWTOOTH WAVE ON VOICE 3
```

SPC

Format: **SPC**(columns)

Usage: Skips **columns**.

The effect is similar to pressing → **<column>** times.

Remarks: The name of this function is derived from **SPACES**, which is misleading. The function prints **cursor right characters**, not spaces. The contents of those character cells that are skipped will not be changed.

Example: Using **SPC**

```
10 FOR I=8 TO 12
20 PRINT SPC(-(I<10));I :REM TRUE = -1, FALSE = 0
30 NEXT I
RUN
 8
 9
10
11
12
```

SPEED

Format: **SPEED** [speed]

Usage: Set CPU clock to 1MHz, 3.5MHz or 40MHz.

speed CPU clock speed where:

- **1** sets CPU to 1MHz.

- **3** sets CPU to 3MHz.
- Anything other than **1** or **3** sets the CPU to 40MHz.

Remarks: Although it's possible to call **SPEED** with any real number, the precision part (the decimal point and any digits after it), will be ignored.

SPEED is a synonym of **FAST**.

SPEED has no effect if `POKE 0,65` has previously been used to set the CPU to 40MHz.

Example: Using **SPEED**

```
10 SPEED      :REM SET SPEED TO MAXIMUM (40 MHZ)
20 SPEED 1    :REM SET SPEED TO 1 MHZ
30 SPEED 3    :REM SET SPEED TO 3.5 MHZ
40 SPEED 3.5  :REM SET SPEED TO 3.5 MHZ
```

SPRCOLOR

Format: **SPRCOLOR** [{mc1**,** mc2}]

Usage: Sets multi-colour sprite colours.

SPRITE, which sets the attributes of a sprite, only sets the foreground colour. For setting the additional two colours of multi-colour sprites, use **SPRCOLOR** instead.

Remarks: The colours used with **SPRCOLOR** will affect all sprites. Refer to the **SPRITE** command for more information.

Example: Using **SPRCOLOR**:

```
10 SPRITE 1,1,2,,,,1  :REM TURN SPRITE 1 ON (FG = 2)
20 SPRCOLOR 4,5       :REM MC1 = 4, MC2 = 5
```

SPRITE

Format: **SPRITE CLR**
SPRITE LOAD filename [**,D** drive] [**,U** unit]
SPRITE SAVE filename [**,D** drive] [**,U** unit]
SPRITE num [{**,** switch**,** colour**,** prio**,** expx**,** expy**,** mode}]

Usage: **SPRITE CLR** clears all sprite data and sets all pointers and attributes to their default values.

SPRITE LOAD loads sprite data from **filename** to sprite memory.

SPRITE SAVE saves sprite data from sprite memory to **filename**.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

The last form switches a sprite on or off and sets its attributes:

num sprite number

switch 1:on, 0:off

colour sprite foreground colour

prio sprite (1) or screen (0) priority

expx 1:sprite X expansion

expy 1:sprite Y expansion

mode 1:multi-colour sprite

Remarks: **SPRCOLOR** must be used to set additional colours for multi-colour sprites (mode = 1).

Example: Using **SPRITE**:

```
2290 CLR:SCNCLR:SPRITE CLR
2300 SPRITE LOAD "DEMOSPRITES1"
2320 FORI=0TO7: C=I: IFC=6THENC=8
2330 MOVSPR I, 60+30*I,0 TO 60+30*I,65+20*I, 3:SPRITE I,1,C,,1,1:NEXT: SLEEP3
2340 FORI=0TO7: SPRITE I,,,,0,0 :NEXT: SLEEP3: SPRITE CLR
2350 FORI=0TO7: MOVSPR I,45*I#5 :NEXT: FORI=0TO7: SPRITE I,1: NEXT
2360 FORI=0TO7:X=60+30*I:Y=65+20*I:DO
2370 LOOPUNTIL(X=RSPPOS(I,.))AND(Y=RSPPOS(I,1)):MOVSPRI,.#.:NEXT
```

SPRSAV

Format: **SPRSAV** source**,** destination

Usage: Copies sprite data.

source sprite number or string variable.

destination sprite number or string variable.

Remarks: Source and destination can either be a sprite number or a string variable, but both cannot be a string variable at the same time. A simple string assignment can be used for such cases.

SPRSAV can be used with the basic form of sprites (C64 compatible) only. These sprites have a size of 64 bytes, and the amount of memory they consume is 67 bytes.

Extended sprites and variable height sprites cannot be used with **SPRSAV**.

Example: Using **SPRSAV**:

```
10 BLOAD "SPRITEDATA",P1600    :REM LOAD DATA FOR SPRITE 1
20 SPRITE 1,1                  :REM TURN SPRITE 1 ON
30 SPRSAV 1,2                  :REM COPY SPRITE 1 DATA TO 2
40 SPRITE 2,1                  :REM TURN SPRITE 2 ON
50 SPRSAV 1,A$                 :REM SAVE SPRITE 1 DATA IN STRING
```

SQR

Format: **SQR**(numeric expression)

Usage: Returns the square root of the numeric expression.

Remarks: The argument must not be negative.

Example: Using **SQR**

```
 PRINT SQR(2)
  1.41421356
```

ST

Format: **ST**

Usage: **ST** holds the status of the last I/O operation. If **ST** is zero, there was no error, otherwise it is set to a device dependent error code.

Remarks: **ST** is a reserved system variable.

Example: Using **ST**

```
100 MX=100:DIM T$(MX)          :REM DATA ARRAY
110 DOPEN#1,"DATA"             :REM OPEN FILE
120 IF DS THEN PRINT"COULD NOT OPEN":STOP
130 LINE INPUT#1,T$(N):N=N+1   :REM READ ONE RECORD
140 IF N>MX THEN PRINT "TOO MANY DATA":GOTO 160
150 IF ST=0 THEN 130           :REM ST = 64 FOR END-OF-FILE
160 DCLOSE#1
170 PRINT "READ";N;" RECORDS"
```

STEP

Format: **FOR** index = start **TO** end [**STEP** step] ... **NEXT** [index]

Usage: **STEP** is an optional part of a **FOR** loop.

The **index** variable may be incremented or decremented by a constant value after each iteration. The default is to increment the variable by 1. The index variable must be a real variable.

start initial value of the index.

end is checked at the end of an iteration, and determines whether another iteration will be performed, or if the loop will exit.

step defines the change applied to to the **index** at the end of a loop iteration. Positive step values increment it, while negative values decrement it. It defaults to 1.0 if not specified.

Remarks: For positive increments, **end** must be greater than or equal to **start**. For negative increments, **end** must be less than or equal to **start**.

It is bad programming practice to change the value of the **index** variable inside the loop or to jump into or out of a loop body with **GOTO**.

Example: Using **STEP**

```
10 FOR D=0 TO 360 STEP 30
20 R = D * π / 180
30 PRINT D;R;SIN(R);COS(R);TAN(R)
40 NEXT D
```

STOP

Format: **STOP**

Usage: Stops the execution of the BASIC program. A message will be displayed showing the line number where the program stopped. The READY. prompt appears and the computer goes into direct mode, waiting for keyboard input.

Remarks: All variable definitions are still valid after **STOP**. They may be inspected or altered, and the program may be continued with **CONT**. However, any editing of the program source will disallow any further continuation.

Program execution can be resumed with **CONT**.

Example: Using **STOP**

```
10 IF V < 0 THEN STOP : REM NEGATIVE NUMBERS STOP THE PROGRAM
20 PRINT SQR(V)       : REM PRINT SQUARE ROOT
```

STR$

Format: **STR$**(numeric expression)

Usage: Returns a string containing the formatted value of the argument, as if it were **PRINT**ed to the string.

Example: Using **STR$**:

```
A$ = "THE VALUE OF PI IS " + STR$(π)
PRINT A$
THE VALUE OF PI IS 3.14159265
```

SYS

Format: **SYS** address [{, areg, xreg, yreg, zreg, sreg}]

Usage: Calls a machine language subroutine. This can be a ROM-resident kernal routine, a BASIC subroutine, or any other routine which has previously been loaded or **POKE**d to RAM.

The CPU registers are loaded with the arguments (if they're specified), then a subroutine call (**JSR address**) is performed. **JSR** is an assembly language instruction that is short for **J**ump to **S**ub**R**outine. The called routine should exit with an **RTS** instruction. **RTS** is another assembly language instruction that is short for **R**e**t**urn from **S**ubroutine. After the subroutine has returned, the register contents will be saved, and the execution of the BASIC program will continue.

address start address of the subroutine.

areg CPU accumulator value.

xreg CPU X register value.

yreg CPU Y register value.

zreg CPU Z register value.

sreg Status register value.

Remarks: The register values after a **SYS** call are stored in system memory. **RREG** can be used to retrieve these values.

SYS uses the current bank, which has been set with **BANK**.

The **SYS** instruction on the MEGA65 is completely different to the well known **SYS** command on the C64. It is not possible to jump to an address after the BASIC program text in bank 0 (and execute a machine language subroutine there), as bank 0 is 64K of RAM with no I/O or kernal mapped in.

Using **SYS** properly (i.e. without corrupting BASIC RAM, and having access to kernal routines and I/O) requires some technical skill, which is out of scope of the User's Guide. However, if you would like to learn more, there is a lot more information and examples in the MEGA65 Developer's Guide.

Example: Using **SYS**:

```
10 BANK 128
20 BLOAD "ML PROG",8192
30 SYS 8192
40 RREG A,X,Y,Z,S
50 PRINT "REGISTER:";A;X;Y;Z;S
```

TAB

Format: **TAB**(column)

Usage: Positions the cursor at **column**.
This is only done if the target column is *right* of the current cursor column, otherwise the cursor will not move. The column count starts with 0 being the left-most column.

Remarks: This function shouldn't be confused with

, which advances the cursor to the next tab-stop.

Example: Using **TAB**

```
10 FOR I=1 TO 5
20 READ A$
30 PRINT "* " A$ TAB(10) " *"
40 NEXT I
50 END
60 DATA ONE,TWO,THREE,FOUR,FIVE

RUN
* ONE      *
* TWO      *
* THREE    *
* FOUR     *
* FIVE     *
```

TAN

Format: **TAN**(numeric expression)

Usage: Returns the tangent of the argument. The argument is expected in units of radians. The result is in the range (-1.0 to +1.0)

Remarks: An argument in units of degrees can be converted to radians by multiplying it with $\pi/180$.

Example: Using **TAN**

```
PRINT TAN(0.7)
 .84228838

X=45:PRINT TAN(X * π / 180)
 .999999999
```

TEMPO

Format: **TEMPO** speed

Usage: Sets the playback speed for **PLAY**.

speed 1-255.

The duration (in seconds) of a whole note is computed with $duration = 24/speed$.

Example: Using **TEMPO**

```
10 VOL 8
20 FOR T = 24 TO 18 STEP -2
30 TEMPO T
40 PLAY "T0M3O4QGAGFED","T2O4M5P0H.DP5GB","T5O3IGAGAGAABABAB"
50 IF RPLAY(1) THEN GOTO 50
60 NEXT T
70 PLAY "T0O5QCO4GEH.C","T2O5IEFEDEDCEGO6P8CP0R","T5O3ICDCDEFEDCO4C"
```

THEN

Format: **IF** expression **THEN** true clause [**ELSE** false clause]

Usage: **THEN** is part of an **IF** statement.

expression is a logical or numeric expression. A numeric expression is evaluated as **FALSE** if the value is zero and **TRUE** for any non-zero value.

true clause one or more statements starting directly after **THEN** on the same line. A line number after **THEN** performs a **GOTO** to that line instead.

false clause one or more statements starting directly after **ELSE** on the same line. A linenumber after **ELSE** performs a **GOTO** to that line instead.

Remarks: The standard **IF ... THEN ... ELSE** structure is restricted to a single line. But the **true clause** and **false clause** may be expanded to several lines using a compound statement surrounded with **BEGIN** and **BEND**.

Example: Using **THEN**

```
1 REM THEN
10 RED$=CHR$(28) : BLACK$=CHR$(144) : WHITE$=CHR$(5)
20 INPUT "ENTER A NUMBER";V
30 IF V<0 THEN PRINT RED$; : ELSE PRINT BLACK$;
40 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
50 PRINT WHITE$
60 INPUT "END PROGRAM: (Y/N)"; A$
70 IF A$="Y" THEN END
80 IF A$="N" THEN 20 : ELSE 60
```

TI

Format: **TI**

Usage: **TI** is a high precision timer with a resolution of 1 micro second.

It is started or reset with **CLR TI**, and can be accessed in the same way as any other variable in expressions.

Remarks: **TI** is a reserved system variable. The value in **TI** is the number of seconds (to 6 decimal places) since it was last cleared or started.

Example: Using **TI**

```
100 CLR TI                :REM START TIMER
110 FOR I%=1 TO 10000:NEXT :REM DO SOMETHING
120 ET = TI               :REM STORE ELAPSED TIME IN ET
130 PRINT "EXECUTION TIME:";ET;" SECONDS"
```

TI$

Format: **TI$**

Usage: **TI$** stores the time information of the RTC (Real-Time Clock) in text form, using the format: "hh:mm:ss". It is updated with every use.

TI$ is a read-only variable, which reads the registers of the RTC and formats the values to a string.

Remarks: **TI$** is a reserved system variable.

It is possible to access the RTC registers directly via **PEEK**. The start address of the registers is at $FFD7110.

For more information on how to set the Real-Time Clock, refer to the Configuration Utility section on page 31.

```
100 REM ****** READ RTC ****** ALL VALUES ARE BCD ENCODED
110 RT = $FFD7110             :REM ADDRESS OF RTC
120 FOR I=0 TO 5              :REM SS,MM,HH,DD,MO,YY
130 T(I)=PEEK(RT+I)           :REM READ REGISTERS
140 NEXT I                    :REM USE ONLY LAST TWO DIGITS
150 T(2) = T(2) AND 127       :REM REMOVE 24H MODE FLAG
160 T(5) = T(5) + $2000       :REM ADD YEAR 2000
170 FOR I=2 TO 0 STEP -1      :REM TIME INFO
180 PRINT USING ">## ";HEX$(T(I));
190 NEXT I
RUN
12 52 36
```

Example: Using **TI$**

```
PRINT DT$;TI$
05-APR-2021 15:10:00
```

TO

Format: keyword **TO**

Usage: **TO** is a secondary keyword used in combination with primary keywords, such as **BACKUP**, **BSAVE**, **CHANGE**, **CONCAT**, **COPY**, **FOR**, **GO**, **RENAME**, and **SET DISK**

Remarks: **TO** cannot be used on its own.

Example: Using **TO**

```
10 GO TO 1000    :REM AS GOTO 1000
20 GOTO 1000     :REM SHORTER AND FASTER
30 FOR I=1 TO 10 :REM TO IS PART OF THE LOOP
40 PRINT I:NEXT  :REM LOOP END
50 COPY "CODES" TO "BACKUP" :REM COPY SINGLE FILE
```

TRAP

Format: **TRAP** [line number]

Usage: **TRAP** with a valid line number registers the BASIC error handler. When a program has an error handler, the run-time behaviour changes. Normally, BASIC will exit the program and display an error message.

However, if a BASIC error handler has been registered, BASIC will instead save the execution pointer and line number, place the error number into the system variable **ER**, and **GOTO** the line number of **TRAP**. The trapping routine can examine **ER** and process the error. From this, the **TRAP** error handler can then decide whether to **STOP** or **RESUME** execution.

TRAP with no argument disables the error handler, and errors will then be handled by the normal system routines.

Example: Using **TRAP**

```
 10 TRAP 100
 20 FOR I=1 TO 100
 30 PRINT EXP(I)
 40 NEXT
 50 PRINT "STOPPED FOR I =";I
 60 END
100 PRINT ERR$(ER): RESUME 50
```

TROFF

Format: **TROFF**

Usage: Turns off trace mode (switched on by **TRON**).

Example: Using **TROFF**

```
 10 TRON            :REM ACTIVATE TRACE MODE
 20 FOR I=85 TO 100
 30 PRINT I;EXP(I)
 40 NEXT
 50 TROFF           :REM DEACTIVATE TRACE MODE

RUN
[10][20][30] 85  8.22301268E+36
[40][30] 86  2.2352466E+37
[40][30] 87  6.0760302E+37
[40][30] 88  1.65163625E+38
[40][30] 89
?OVERFLOW ERROR IN 30
READY.
```

TRON

Format: **TRON**

Usage: Turns on trace mode.

Example: Using **TRON**

```
 10 TRON            :REM ACTIVATE TRACE MODE
 20 FOR I=85 TO 100
 30 PRINT I;EXP(I)
 40 NEXT
 50 TROFF           :REM DEACTIVATE TRACE MODE

RUN
[10][20][30] 85  8.22301268E+36
[40][30] 86  2.2352466E+37
[40][30] 87  6.0760302E+37
[40][30] 88  1.65163625E+38
[40][30] 89
?OVERFLOW ERROR IN 30
READY.
```

TYPE

Format: **TYPE** filename [**,D** drive] [**,U** unit]

Usage: Prints the contents of a file containing text encoded as PETSCII.

filename is either a quoted string such as "DATA", or a string expression in brackets such as (FI$).

drive drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units such as the 1541, 1571, or 1581.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **TYPE** cannot be used to print BASIC programs. Use **LIST** for programs instead. **TYPE** can only process **SEQ** or **USR** files containing records of PETSCII text, delimited by the CR character.

The CR character is also knows as carriage return, and can be created by using **CHR$(13)** .

Example: Using **TYPE**

```
TYPE "README"
TYPE "README 1ST",U9
```

UNTIL

Format: **DO** ... **LOOP**
DO [<**UNTIL** | **WHILE**> logical expression]
. . . statements [**EXIT**]
LOOP [<**UNTIL** | **WHILE**> logical expression]

Usage: **DO** and **LOOP** define the start of a BASIC loop. Using **DO** and **LOOP** alone without any modifiers creates an infinite loop, which can only be exited by the **EXIT** statement. The loop can be controlled by adding **UNTIL** or **WHILE** after the **DO** or **LOOP**.

Remarks: **DO** loops may be nested. An **EXIT** statement exits the current loop only.

Examples: Using **DO** and **LOOP**.

```
10 PW$="":DO
20 GET A$:PW$=PW$+A$
30 LOOP UNTIL LEN(PW$))7 OR A$=CHR$(13)

10 DO : REM WAIT FOR USER DECISION
20 GET A$
30 LOOP UNTIL A$="Y" OR A$="N" OR A$="y" OR A$="n"

10 DO WHILE ABS(EPS) ) 0.001
20 GOSUB 2000 : REM ITERATION SUBROUTINE
30 LOOP

10 I%=0 : REM INTEGER LOOP 1-100
20 DO I%=I%+1
30 LOOP WHILE I% ( 101
```

USING

Format: **PRINT**[**#** channel**,**] **USING** format**;** argument

Usage: Parses the **format** string and evaluates the argument. The argument can be either a string or a numeric value. The format of the resulting output is directed by the **format** string.

channel number, which was given to a previous call to commands such as **APPEND**, **DOPEN**, or **OPEN**. If no channel is specified, the output goes to the screen.

format string variable or a string constant which defines the rules for formatting. When using a number as the **argument**, formatting can

be done in either CBM style, providing a pattern such as ####.## or in C style using a <width.precision> specifier, such as %3D %7.2F %4X .

argument the number to be formatted. If the argument does not fit into the format e.g. trying to print a 4 digit variable into a series of three hashes (###), asterisks will be used instead.

Remarks: The format string is only applied for one argument, but it is possible to append more than one **USING format;argument** sequences.

argument may consist of printable characters and control codes. Printable characters are printed to the cursor position, while control codes are executed. The number of # characters sets the width of the output. If the first character of the format string is an equals '=' sign, the argument string is centered. If the first character of the format string is a greater than '>' sign, the argument string is right justified.

Example: **USING** with a corresponding **PRINT#**

```
PRINT USING "##.##";π, USING " [%6.4F] ";SQR(2)
 3.14 [1.4142]

PRINT USING " ( # # # ) ";12*31
 ( 3 7 2 )

PRINT USING "###"; "ABCDE"
ABC

PRINT USING ">###"; "ABCDE"
CDE

PRINT USING "ADDRESS:$%4X";65000
ADDRESS:$FDE8

A$="####,####,####.#":PRINT USING A$;1E8/3
 33,333,333.3
```

USR

Format: **USR**(numeric expression)

Usage: Invokes an assembly language routine whose memory address is stored at $02F8 - $02F9. The result of the **numeric expression** is written to floating point accumulator 1.

After executing the assembly routine, BASIC returns the contents of the floating point accumulator 1.

Remarks: Banks 0-127 give access to RAM or ROM banks. Banks greater than 127 are used to access I/O, and the underlying system hardware such as the VIC, SID, FDC, etc.

If you would like to learn more, there is a lot more information and examples in the MEGA65 Developer's Guide.

Example: Using **USR**

```
10 UX = DEC("7F00")            :REM ADDRESS OF USER ROUTINE
20 BANK 128                    :REM SELECT SYSTEM BANK
30 BLOAD "ML-PROG",P(UX)       :REM LOAD USER ROUTINE
40 POKE (DEC("2F8")),UX AND 255 :REM USR JUMP TARGET LOW
50 POKE (DEC("2F9")),UX / 256   :REM USR JUMP TARGET HIGH
60 PRINT USR(π)                :REM PRINT RESULT FOR ARGUMENT PI
```

VAL

Format: **VAL**(string expression)

Usage: Converts a string to a floating point value.

This function acts in the same way as reading from a string.

Remarks: A string containing an invalid number will not produce an error, but return 0 as the result instead.

Example: Using **VAL**

```
PRINT VAL("78E2")
7800

PRINT VAL("7+5")
7

PRINT VAL("1.256")
1.256

PRINT VAL("$FFFF")
0
```

VERIFY

Format: **VERIFY** filename [**,** unit [**,** binflag]]

Usage: **VERIFY** with no **binflag** compares a BASIC program in memory with a disk file of type **PRG**. It does the same as **DVERIFY**, but the syntax is different.

VERIFY with **binflag** compares a binary file in memory with a disk file of type **PRG**. It does the same as **BVERIFY**, but the syntax is different.

filename is either a quoted string, e.g. `"PROG"` or a string expression.

unit device number on the IEC bus. Typically in the range from 8 to 11 for disk units. If a variable is used, it must be placed in brackets. The unit # defaults to **8**.

Remarks: **VERIFY** can only test for equality. It gives no information about the number or position of different valued bytes. **VERIFY** exits with either the message `OK` or with `VERIFY ERROR`.

VERIFY is obsolete in BASIC 65. It is only here for backwards compatibility. It is recommended to use **DVERIFY** and **BVERIFY** instead.

Examples: Using **VERIFY**

```
VERIFY "ADVENTURE"
VERIFY "ZORK-I",9
VERIFY "1:DUNGEON",10
```

VIEWPORT

Format: **VIEWPORT CLR**
VIEWPORT DEF x, y, dx, dy

Usage: **VIEWPORT DEF** defines a clipping region with the origin (upper left position) set to **x, y** and the width **dx** and the height **dy**. All following graphics commands are limited to the **VIEWPORT** region.

Remarks: **VIEWPORT** must be followed by either **CLR**, or **DEF** with four integer parameters. **VIEWPORT CLR** resets the clipping region to the entire screen.

Info: **VIEWPORT** wasn't completed in BASIC 10, and a working implementation will be available in a future BASIC 65 update.

Example: Using **VIEWPORT**

```
VIEWPORT DEF 20,20,100,100 :REM REGION 20->119, 20->119
VIEWPORT CLR               :REM FULL SCREEN
```

VOL

Format: **VOL** volume

Usage: Sets the volume for sound output with **SOUND** or **PLAY**.

volume 0 (off) to 15 (loudest).

Remarks: This volume setting affects all voices.

Example: Using **VOL**

```
10 TEMPO 22
20 FOR V = 2 TO 8 STEP 2
30 VOL V
40 PLAY "T0M3O4QGAGFED","T2O4M5P0H.DP5GB","T5O3IGAGAGAABABAB"
50 IF RPLAY(1) THEN GOTO 50
60 NEXT V
70 PLAY "T005QCO4GEH.C","T2O5IEFEDEDCEGOGP9CP0R","T5O3ICDCDEFEDCO4C"
```

WAIT

Format: **WAIT** address**,** andmask [**,** xormask]

Usage: Pauses the BASIC program until a requested bit pattern is read from the given address.

address the address at the current memory bank, which is read.

andmask AND mask applied.

xormask XOR mask applied.

WAIT reads the byte value from **address** and applies the masks: **result = PEEK(address) AND andmask XOR xormask**.

The pause ends if the result is non-zero, otherwise reading is repeated. This may hang the computer indefinitely if the condition is never met.

Remarks: **WAIT** is typically used to examine hardware registers or system variables and wait for an event, e.g. joystick event, mouse event, keyboard press or a specific raster line is about to be drawn to the screen.

Example: Using **WAIT**

```
10 BANK 128
20 WAIT 211,1          :REM WAIT FOR SHIFT KEY BEING PRESSED
```

WHILE

Format: **DO** ... **LOOP**
DO [<**UNTIL** | **WHILE**> logical expression]
. . . statements [**EXIT**]
LOOP [<**UNTIL** | **WHILE**> logical expression]

Usage: **DO** and **LOOP** define the start of a BASIC loop. Using **DO** and **LOOP** alone without any modifiers creates an infinite loop, which can only be exited by the **EXIT** statement. The loop can be controlled by adding **UNTIL** or **WHILE** after the **DO** or **LOOP**.

Remarks: **DO** loops may be nested. An **EXIT** statement exits the current loop only.

Examples: Using **DO** and **LOOP**

```
10 PW$="":DO
20 GET A$:PW$=PW$+A$
30 LOOP UNTIL LEN(PW$)>7 OR A$=CHR$(13)

10 DO : REM WAIT FOR USER DECISION
20 GET A$
30 LOOP UNTIL A$="Y" OR A$="N" OR A$="y" OR A$="n"

10 DO WHILE ABS(EPS) > 0.001
20 GOSUB 2000 : REM ITERATION SUBROUTINE
30 LOOP

10 I%=0 : REM INTEGER LOOP 1-100
20 DO I%=I%+1
30 LOOP WHILE I% < 101
```

WINDOW

Format: **WINDOW** left, top, right, bottom [, clear]

Usage: Sets the text screen window.

left left column

top top row

right right column

bottom bottom row

clear clear text window flag

Remarks: The row values range from 0 to 24. The column values range from 0 to either 39 or 79. This depends on the screen mode.

There can be only one window on the screen. Pressing CLR HOME twice or **PRINT**ing **CHR$(19)CHR$(19)** will reset the window to the default (full screen).

Example: Using **WINDOW**

```
10 WINDOW 0,1,79,24     :REM SCREEN WITHOUT TOP ROW
20 WINDOW 0,0,79,24,1   :REM FULL SCREEN WINDOW CLEARED
30 WINDOW 0,12,79,24    :REM LOWER HALF OF SCREEN
40 WINDOW 20,5,59,15    :REM SMALL CENTRED WINDOW
```

XOR

Format: operand **XOR** operand

Usage: The Boolean **XOR** operator performs a bit-wise logical exclusive OR operation on two 16-bit values. Integer operands are used as they are. Real operands are converted to a signed 16-bit integer (losing precision). Logical operands are converted to 16-bit integer using $FFFF, (decimal -1) for TRUE, and $0000 (decimal 0) for FALSE.

Expression	Result
0 XOR 0	0
0 XOR 1	1
1 XOR 0	1
1 XOR 1	0

Remarks: The result is of type integer. If the result is used in a logical context, the value of 0 is regarded as FALSE, and all other non-zero values are regarded as TRUE.

Example: Using **XOR**

```
FOR I = 0 TO 8: PRINT I XOR 5;: NEXT I
 5  4  7  6  1  0  3  2  13
```

APPENDIX B

Special Keyboard Controls and Sequences

- PETSCII Codes and CHR$
- Control codes
- Shifted codes
- Escape Sequences

PETSCII CODES AND CHR$

In BASIC, `PRINT CHR$(X)` can be used to print a character from a PETSCII code. Below is the full table of PETSCII codes you can print by index. For example, while in the default uppercase/graphics mode, by using index 65 from the table below as: `PRINT CHR$(65)` you will print the letter `A`. You can read more about **CHR$** on page 92.

You can also do the reverse with the ASC statement. For example: `PRINT ASC("A")` will output `65`, which matches the code in the table.

Note: Function key (F1-F14 + HELP) values in this table are not intended to be printed via `CHR$()`, but rather to allow function-key input to be assessed in BASIC programs via the GET / GETKEY commands.

0

1

2 UNDERLINE ON

3

4

5 WHITE

6

7 BELL

8

9 TAB

10 LINEFEED

11 DISABLE SHIFT

12 ENABLE SHIFT

13 RETURN

14 LOWER CASE

15 BLINK/FLASH ON

16 F9

17 ↓

18 RVS ON

19 CLR HOME

20 INST DEL

21 F10 / BACK WORD

22 F11

23 F12 / NEXT WORD

24 SET/CLEAR TAB

25 F13

26 F14 / BACK TAB

27 ESCAPE

28 RED

29 →

30 GREEN

31 BLUE

32 SPACE

33 !

34 ″

35 #

36 $

37 %

38 &

39 ′

40 (

41)

42 *

43 +

44 ,

45 -

46 .

47 /

48 0

49 1

50 2

51 3

52 4

53 5

54 6

55 7

56 8

57 9

58 :

59 ;

60 <

61 =

62 >

63 ?

64 @

65 A

66 B

67 C

68 D

69 E

70 F

71 G

72 H

73 I

74 J

75 K

76 L

77 M

78 N

79 O

80 P

81 Q

82 R

83 S

84 T

85 U

86 V

87 W
88 X
89 Y
90 Z
91 [
92 £
93]
94 ↑
95 ←
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126 π
127
128
129 ORANGE
130 UNDERLINE OFF
131
132 HELP
133 F1
134 F3
135 F5
136 F7
137 F2
138 F4
139 F6
140 F8
141 SHIFT RETURN
142 UPPERCASE
143 BLINK/FLASH OFF
144 BLACK
145 ↑
146 RVS OFF
147 SHIFT CLR HOME
148 SHIFT INST DEL
149 BROWN
150 LT. RED
151 DK. GRAY
152 GRAY
153 LT. GREEN
154 LT. BLUE
155 LT. GRAY
156 PURPLE
157 ←
158 YELLOW
159 CYAN
160 SPACE
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191

Note 1: Codes from 192 to 223 are the equal to 96 to 127. Codes from 224 to 254 are equal to 160 to 190, and code 255 is equal to 126.

Note 2: While using lowercase/uppercase mode (by pressing C= + SHIFT), be aware that:

- The uppercase letters in region 65-90 of the above table are replaced with lowercase letters.
- The graphical characters in region 97-122 of the above table are replaced with uppercase letters.
- PETSCII's lowercase (65-90) and uppercase (97-122) letters are in ASCII's uppercase (65-90) and lowercase (97-122) letter regions.

CONTROL CODES

Keyboard Control	Function
Colours	
CTRL + 1 to 8	Choose from the first range of colours. More information on the colours available is under the BASIC **BACKGROUND** command on page 79.
C= + 1 to 8	Choose from the second range of colours.
CTRL + E	Restores the colour of the cursor back to the default (white).
CTRL + D	Switches the VIC-IV to colour range 0-15 (default colours). These colours can be accessed with CTRL and keys 1 to 8 (for the first 8 colours), or C= and keys 1 to 8 (for the remaining 8 colours).
CTRL + A	Switches the VIC-IV to colour range 16-31 (alternate/rainbow colours). These colours can be accessed with CTRL and keys 1 to 8 (for the first 8 colours), or C= and keys 1 to 8 (for the remaining 8 colours).
Tabs	

Keyboard Control	Function
CTRL + Z	Tabs the cursor to the left. If there are no tab positions remaining, the cursor will remain at the start of the line.
CTRL + I	Tabs the cursor to the right. If there are no tab positions remaining, the cursor will remain at the end of the line.
CTRL + X	Sets or clears the current screen column as a tab position. Use CTRL + Z and I to jump back and forth to all positions set with X.
Movement	
CTRL + Q	Moves the cursor down one line at a time. Equivalent to ↓.
CTRL + J	Moves the cursor down a position. If you are on a long line of BASIC code that has extended to two lines, then the cursor will move down two rows to be on the next line.
CTRL +]	Equivalent to →.
CTRL + T	Backspace the character immediately to the left and to shift all rightmost characters one position to the left. This is equivalent to INST DEL.
CTRL + M	Performs a carriage return, equivalent to RETURN.
Word movement	
CTRL + U	Moves the cursor back to the start of the previous word. If there are no words between the current cursor position and the start of the line, the cursor will move to the first column of the current line.

Keyboard Control	Function
CTRL + W	Advances the cursor forward to the start of the next word. If there are no words between the cursor and the end of the line, the cursor will move to the first column of the next line.
Scrolling	
CTRL + P	Scroll BASIC listing down one line. Equivalent to F9.
CTRL + V	Scroll BASIC listing up one line. Equivalent to F11.
CTRL + S	Equivalent to NO SCROLL.
Formatting	
CTRL + B	Enables underline text mode. You can disable underline mode by pressing ESC, then O.
CTRL + O	Enables flashing text mode. You can disable flashing mode by pressing ESC, then O.
Casing	
CTRL + N	Changes the text case mode from uppercase to lowercase.
CTRL + K	Locks the uppercase/lowercase mode switch usually performed with + SHIFT.
CTRL + L	Enables the uppercase/lowercase mode switch that is performed with the + SHIFT.
Miscellaneous	
CTRL + G	Produces a bell tone.
CTRL + [	Equivalent to pressing ESC.
CTRL + *	Enters the Matrix Mode Debugger.

SHIFTED CODES

Keyboard Control	Function
SHIFT + INST DEL	Insert a character at the current cursor position and move all characters to the right by one position.
SHIFT + HOME	Clear home, clear the entire screen, and move the cursor to the home position.

ESCAPE SEQUENCES

To perform an Escape Sequence, press and release ESC, then press one of the following keys to perform the sequence:

Key	Sequence
Editor behaviour	
ESC X	Clears the screen and toggles between 40 and 80-column modes.
ESC @	Clears a region of the screen, starting from the current cursor position, to the end of the screen.
ESC O	Cancels the quote, reverse, underline, and flash modes.
Scrolling	
ESC V	Scrolls the entire screen up one line.
ESC W	Scrolls the entire screen down one line.

Key	Sequence
ESC L	Enables scrolling when ↓ is pressed at the bottom of the screen.
ESC M	Disables scrolling. When pressing ↓ at the bottom of the screen, the cursor will move to the top of the screen. However, when pressing ↑ at the top of the screen, the cursor will remain on the first line.
Insertion and deletion	
ESC I	Inserts an empty line at the current cursor position and moves all subsequent lines down one position.
ESC D	Deletes the current line and moves lines below the cursor up one position.
ESC P	Erases all characters from the cursor to the start of the current line.
ESC Q	Erases all characters from the cursor to the end of the current line.
Movement	
ESC J	Moves the cursor to the start of the current line.
ESC K	Moves the cursor to the last non-whitespace character on the current line.
ESC ↑	Saves the current cursor position. Use ESC ← (next to 1) to move it back to the saved position. Note that the ↑ used here is next to RESTORE.
ESC ←	Restores the cursor position to the position stored via a prior a press of the ESC ↑ (next to RESTORE) key sequence. Note that the ← used here is next to 1.

Key	Sequence
Windowing	
ESC T	Sets the top-left corner of the windowed area. All typed characters and screen activity will be restricted to the area. Also see ESC B. Windowed mode can be disabled by pressing CLR HOME twice.
ESC B	Sets the bottom right corner of the windowed area. All typed characters and screen activity will be restricted to the area. Also see ESC T. Windowed mode can be disabled by pressing CLR HOME twice.
Cursor behaviour	
ESC A	Enables auto-insert mode. Any keys pressed will be inserted at the current cursor position, shifting all characters on the current line after the cursor to the right by one position.
ESC C	Disables auto-insert mode, reverting back to overwrite mode.
ESC E	Sets the cursor to non-flashing mode.
ESC F	Sets the cursor to regular flashing mode.
Bell behaviour	
ESC G	Enables the bell which can be sounded using CTRL and G.
ESC H	Disable the bell so that pressing CTRL and G will have no effect.
Colours	

Key	Sequence
ESC U	Switches the VIC-IV to colour range 0-15 (default colours). These colours can be accessed with CTRL and keys 1 to 8 (for the first 8 colours), or M and keys 1 to 8 (for the remaining colours).
ESC S	Switches the VIC-IV to colour range 16-31 (alternate/rainbow colours). These colours can be accessed with CTRL and keys 1 to 8 (for the first 8 colours), or M and keys 1 to 8 (for the remaining colours).
Tabs	
ESC Y	Set the default tab stops (every 8 spaces) for the entire screen.
ESC Z	Clears all tab stops. Any tabbing with CTRL and I will move the cursor to the end of the line.

APPENDIX C

Supporters & Donors

- Organisations
- Contributors
- Supporters

The MEGA65 would not have been possible to create without the generous support of many organisations and individuals.

We are still compiling these lists, so apologies if we haven't included you yet. If you know anyone we have left out, please let us know, so that we can recognise the contribution of everyone who has made the MEGA65 possible, and into the great retro-computing project that it has become.

ORGANISATIONS

The MEGA Museum of Electronic Games & Art e.V. Germany
EVERYTHING

Trenz Electronik, Germany
MOTHERBOARD

Hintsteiner, Austria
CASE

GMK, Germany
KEYBOARD

CONTRIBUTORS

Andreas Liebeskind
(libi in paradize)
CFO MEGA eV

Gürçe Işıkyıldız
(Gurce)
Tools and enhancements

Russell Peake
(rdpeake)
Bug Herding

Alexander Nik Petra
(n0d)
Early Case Design

Ralph Egas
(0-limits)
Business Advisor

Lucas Moss

MEGAphone PCB Design

Daren Klamer
(Impakt)
Manual proof-reading

Dr. Canan Hastik
(indica)
Chairwoman MEGA eV

Simon Jameson
(Shallan)
Platform Enhancements

Stephan Kleinert
(ubik)
Destroyer of BASIC 10

Wayne Johnson
(sausage)
Manual Additions

Lukas Kleiss
(LAK132)
MegaWAT Presentation Software

Maurice van Gils
(Maurice)
BASIC 65 example programs

SUPPORTERS

@11110110100
3c74ce64
8-Bit Classics
Aaron Smith
Achim Mrotzek
Adolf Nefischer
Adrian Esdaile
Adrien Guichard
Ahmed Kablaoui
Alan Bastian Witkowski
Alan Field
Alastair Paulin-Campbell
Alberto Mercuri
Alexander Haering
Alexander Kaufmann
Alexander Niedermeier
Alexander Soppart
Alfonso Ardire
Amiga On The Lake
André Kudra
André Simeit
André Wösten
Andrea Farolfi
Andrea Minutello
Andreas Behr
Andreas Freier
Andreas Grabski
Andreas Millinger
Andreas Nopper
Andreas Ochs
Andreas Wendel Manufaktur
Andreas Zschunke
Andrew Bingham
Andrew Dixon
Andrew Mondt
Andrzej Hłuchyj
Andrzej Sawiniec
Andrzej Śliwa
Anthony W. Leal
Arkadiusz Bronowicki
Arkadiusz Kwasny
Arnaud Léandre
Arne Drews
Arne Neumann
Arne Richard Tyarks
Axel Klahr
Balaz Ondrej
Barry Thompson
Bartol Filipovic
Benjamin Maas
Bernard Alaiz
Bernhard Zorn
Bieno Marti-Braitmaier
Bigby
Bill LaGrue
Bjoerg Stojalowski
Björn Johannesson
Bjørn Melbøe
Bo Goeran Kvamme
Boerge Noest
Bolko Beutner
Brett Hallen
Brian Gajewski
Brian Green
Brian Juul Nielsen
Brian Reiter
Bryan Pope
Burkhard Franke
Byron Goodman
Cameron Roberton (KONG)
Carl Angervall
Carl Danowski
Carl Stock
Carl Wall
Carlo Pastore
Carlos Silva
Carsten Sørensen
Cenk Miroglu Miroglu
Chang sik Park
Charles A. Hutchins Jr.
Chris Guthrey
Chris Hooper
Chris Stringer
Christian Boettcher
Christian Eick
Christian Gleinser
Christian Gräfe
Christian Heffner
Christian Kersting
Christian Schiller
Christian Streck
Christian Weyer
Christian Wyk
Christoph Haug
Christoph Huck
Christoph Pross
Christopher Christopher
Christopher Kalk
Christopher Kohlert
Christopher Nelson
Christopher Taylor
Christopher Whillock
Claudio Piccinini
Claus Skrepek
Collen Blijenberg
Constantine Lignos
Crnjaninja
Daniel Auger
Daniel Julien
Daniel Lobitz
Daniel O'Connor
Daniel Teicher
Daniel Tootill
Daniel Wedin
Daniele Benetti
Daniele Gaetano Capursi
Dariusz Szczesniak
Darrell Westbury
David Asenjo Raposo
David Dillard
David Gorgon
David Norwood
David Raulo
David Ross
de voughn accooe
Dean Scully
Dennis Jeschke
Dennis Schaffers
Dennis Schierholz

Dennis Schneck
denti
Dick van Ginkel
Diego Barzon
Dierk Schneider
Dietmar Krueger
Dietmar Schinnerl
Dirk Becker
Dirk Wouters
Domingo Fivoli
DonChaos
Donn Lasher
Douglas Johnson
Dr. Leopold Winter
Dusan Sobotka
Earl Woodman
Ed Reilly
Edoardo Auteri
Eduardo Gallardo
Eduardo Luis Arana
Eirik Juliussen Olsen
Emilio Monelli
EP Technical Services
Epic Sound
Erasmus Kuhlmann
ergoGnomik
Eric Hilaire
Eric Hildebrandt
Eric Hill
Eric Jutrzenka
Erwin Reichel
Espen Skog
Evangelos Mpouras
Ewan Curtis
Fabio Zanicotti
Fabrizio Di Dio
Fabrizio Lodi
FARA Gießen GmbH
FeralChild
First Choice Auto's
Florian Rienhardt
Forum64. de
Francesco Baldassarri
Frank Fechner
Frank Glaush
Frank Gulasch
Frank Haaland
Frank Hempel
Frank Koschel
Frank Linhares
Frank Sleeuwaert
Frank Wolf
FranticFreddie
Fredrik Ramsberg
Fridun Nazaradeh
Friedel Kropp
Garrick West
Gary Lake-Schaal
Gary Pearson
Gavin Jones
Geir Sigmund Straume
Gerd Mitlaender
Giampietro Albiero
Giancarlo Valente
Gianluca Girelli
Giovanni Medina
Glen Fraser
Glen R Perye III
Glenn Main
Gordon Rimac
GRANT BYERS
Grant Louth
Gregor Bubek
Gregor Gramlich
Guido Ling
Guido von Gösseln
Guillaume Serge
Gunnar Hemmerling
Günter Hummel
Guy Simmons
Guybrush Threepwood
Hakan Blomqvist
Hans Pronk
Hans-Jörg Nett
Hans-Martin Zedlitz
Harald Dosch
Harri Salokorpi
Harry Culpan
Heath Gallimore
Heinz Roesner
Heinz Stampfli
Helge Förster
Hendrik Fensch
Henning Harperath
Henri Parfait
Henrik Kühn
Holger Burmester
Holger Sturk
Howard Knibbs
Hubert de Hollain
Huberto Kusters
Hugo Maria Gerardus v.d. Aa
Humberto Castaneda
Ian Cross
IDE64 Staff
Igor Ianov
Immo Beutler
Ingo Katte
Ingo Keck
Insanely Interested Publishing
IT-Dienstleistungen Obsieger
Ivan Elwood
Jaap HUIJSMAN
Jace Courville
Jack Wattenhofer
Jakob Schönpflug
Jakub Tyszko
James Hart
James Marshburn
James McClanahan
James Sutcliffe
Jan Bitruff
Jan Hildebrandt
Jan Iemhoff
Jan Kösters
Jan Peter Borsje
Jan Schulze
Jan Stoltenberg-Lerche
Janne Tompuri
Jannis Schulte
Jari Loukasmäki
Jason Smith
Javier Gonzalez Gonzalez
Jean-Paul Lauque
Jeffrey van der Schilden
Jens Schneider
Jens-Uwe Wessling
Jesse DiSimone

Jett Adams
Johan Arneklev
Johan Berntsson
Johan Svensson
Johannes Fitz
John Cook
John Deane
John Dupuis
John Nagi
John Rorland
John Sargeant
John Traeholt
Jon Sandelin
Jonas Bernemann
Jonathan Prosise
Joost Honig
Jordi Pakey-Rodriguez
Jöre Weber
Jörg Jungermann
Jörg Schaeffer
Jörg Weese
Josef Hesse
Josef Soucek
Josef Stohwasser
Joseph Clifford
Joseph Gerth
Jovan Crnjanin
Juan Pablo Schisano
Juan S. Cardona Iguina
JudgeBeeb
Juliussen Olsen
Juna Luis Fernandez Garcia
Jürgen Endras
Jürgen Herm Stapelberg
Jyrki Laurila
Kai Pernau
Kalle Pöyhönen
Karl Lamford
Karl-Heinz Blum
Karsten Engstler
Karsten Westebbe
katarakt
Keith McComb
Kenneth Dyke
Kenneth Joensson
Kevin Edwards
Kevin Thomasson
Kim Jorgensen
Kim Rene Jensen
Kimmo Hamalainen
Konrad Buryło
Kosmas Einbrodt
Kurt Klemm
Lachlan Glaskin
Large bits collider
Lars Becker
Lars Edelmann
Lars Slivsgaard
Lasse Lambrecht
Lau Olivier
Lee Chatt
Loan Leray
Lorenzo Quadri
Lorenzo Travagli
Lorin Millsap
Lothar James Foss
Lothar Serra Mari
Luca Papinutti
Ludek Smetana
Lukas Burger
Lutz-Peter Buchholz
Luuk Spaetgens
Mad Web Skills
MaDCz
Magnus Wiklander
Maik Diekmann
Malte Mundt
Manfred Wittemann
Manuel Beckmann
Manzano Mérida
Marc "3D-vice" Schmitt
Marc Bartel
Marc Jensen
Marc Schmidt
Marc Theunissen
Marc Tutor
Marc Wink
Marcel Buchtmann
Marcel Kante
Marco Beckers
Marco Cappellari
Marco Rivela
Marco van de Water
Marcus Gerards
Marcus Herbert
Marcus Linkert
Marek Pernicky
Mario Esposito
Mario Fetka
Mario Teschke
Mariusz Tymków
Mark Adams
Mark Anderson
Mark Green
Mark Hucker
Mark Leitiger
Mark Spezzano
Mark Watkin
Marko Rizvic
Markus Bieler
Markus Bonet
Markus Dauberschmidt
Markus Fehr
Markus Fuchs
Markus Guenther-Hirn
Markus Liukka
Markus Merz
Markus Roesgen
Markus Uttenweiler
Martin Bauhuber
Martin Benke
Martin Gendera
Martin Groß
Martin Gutenbrunner
Martin Johansen
Martin Marbach
Martin Sonnleitner
Martin Steffen
Marvin Hardy
Massimo Villani
Mathias Dellacherie
Mathieu Chouinard
Matthew Adams
Matthew Browne
Matthew Carnevale
Matthew Palmer
Matthew Santos
Matthias Barthel

Matthias Dolenc
Matthias Fischer
Matthias Frey
Matthias Grandis
Matthias Guth
Matthias Lampe
Matthias Meier
Matthias Mueller
Matthias Nofer
Matthias Schonder
Maurice Al-Khaliedy
Max Ihlenfeldt
Meeso Kim
Michael Dailly
Michael Dötsch
Michael Dreßel
Michael Fichtner
Michael Fong
Michael Geoffrey Stone
Michael Gertner
Michael Grün
Michael Habel
Michael Härtig
Michael Haynes
Michael J Burkett
Michael Jensen
Michael Jurisch
Michael Kappelgaard
Michael Kleinschmidt
Michael Lorenz
Michael Mayerhofer
Michael Nurney
Michael Rasmussen
Michael Richmond
Michael Sachse
Michael Sarbak
Michael Schneider
Michael Scholz
Michael Timm
Michael Traynor
Michael Whipp
Michal Ursiny
Michele Chiti
Michele Perini
Michele Porcu
Miguel Angel Rodriguez Jodar
Mikael Lund
Mike Betz
Mike Kastrantas
Mike Pikowski
Mikko Hämäläinen
Mikko Suontausta
Mirko Roller
Miroslav Karkus
Morgan Antonsson
Moritz
Morten Nielsen
MUBIQUO APPS,SL
Myles Cameron-Smith
Neil Moore
Nelson
neoman
Nicholas Melnick
Nikolaj Brinch Jørgensen
Nils Andreas
Nils Eilers
Nils Hammerich
Nils77
Norah Smith
Norman King
Normen Zoch
Olaf Grunert
Ole Eitels
Oliver Boerner
Oliver Brüggmann
Oliver Graf
Oliver Smith
Olivier Bori
ONEPSI LLC
oRdYNe
Osaühing Trioflex
OSHA-PROS USA
Padawer
Patrick Becher
Patrick Bürckstümmer
Patrick de Zoete
Patrick Toal
Patrick Vogt
Paul Alexander Warren
Paul Gerhardt (KONG)
Paul Jackson
Paul Johnson
Paul Kuhnast (mindrail)
Paul Massay
Paul Westlake
Paul Wögerer
Pauline Brasch
Paulo Apolonia
Pete Collin
Pete of Retrohax.net
Peter Eliades
Peter Gries
Peter Habura
Peter Herklotz
Peter Huyoff
Peter Knörzer
Peter Leswell
Peter Weile
Petri Alvinen
Philip Marien
Philip Timmermann
Philipp Rudin
Pierre Kressmann
Pieter Labie
Piotr Kmiecik
Power-on.at
Przemysław Safonow
Que Labs
R Welbourn
R-Flux
Rafał Michno
Rainer Kappler
Rainer Kopp
Rainer Weninger
Ralf Griewel
Ralf Pöscha
Ralf Reinhardt
Ralf Schenden
Ralf Smolarek
Ralf Zenker
Ralph Bauer
Ralph Wernecke
Rédl Károly
Reiner Lanowski
Remi Veilleux
Riccardo Bianchi
Richard Englert
Richard Good

Richard Menedetter
Richard Sopuch
Rick Reynolds
Rico Gruninger
Rob Dean
Robert Bernardo
Robert Eaglestone
Robert Grasböck
Robert Miles
Robert Schwan
Robert Shively
Robert Tangmar
Robert Trangmar
Rodney Xerri
Roger Olsen
Roger Pugh
Roland Attila Kett
Roland Evers
Roland Schatz
Rolf Hass
Ronald Cooper
Ronald Hunn
Ronny Hamida
Ronny Preiß
Roy van Zundert
Rüdiger Wohlfromm
Ruediger Schlenter
Rutger WIllemsen
Sampo Peltonen
Sarmad Gilani
SAS74
Sascha Hesse
Scott Halman
Scott Hollier
Scott Robison
Sebastian Baranski
Sebastian Bölling
Sebastian Felzmann
Sebastian Lipp
Sebastian Rakel
Şemseddin Moldibi
Seth Morabito
Shawn McKee
Siegfried Hartmann
Sigurbjorn Larusson
Sigurdur Finnsson
Simon Lawrence
Simon Wolf
spreen.digital
Stefan Haberl
Stefan Kramperth
Stefan Richter
Stefan Schultze
Stefan Sonnek
Stefan Theil
Stefan Vrampe
Stefano Canali
Stefano Mozzi
Steffen Reiersen
Stephan Bielmann
Stephen Jones
Stephen Kew
Steve Gray
Steve Kurlin
Steve Lemieux
Steven Combs
Stewart Dunn
Stuart Marsh
Sven Neumann
Sven Stache
Sven Sternberger
Sven Wiegand
Szabolcs Bence
Tantrumedia Limited
Techvana Operations Ltd.
Teddy Turmeaux
Teemu Korvenpää
The Games Foundation
Thierry Supplisson
Thieu-Duy Thai
Thomas Bierschenk
Thomas Edmister
Thomas Frauenknecht
Thomas Gitzen
Thomas Gruber
Thomas Haidler
Thomas Jager
Thomas Karlsen
Thomas Laskowski
Thomas Marschall
Thomas Niemann
Thomas Scheelen
Thomas Schilling
Thomas Tahsin-Bey
Thomas Walter
Thomas Wirtzmann
Thorsten Knoll
Thorsten Nolte
Tim Krome
Tim Waite
Timo Weirich
Timothy Blanks
Timothy Henson
Timothy Prater
Tobias Butter
Tobias Heim
Tobias Köck
Tobias Lüthi
Tommi Vasarainen
Toni Ammer
Tore Olsen
Torleif Strand
Torsten Schröder
Tuan Nguyen
Uffe Jakobsen
Ulrich Hintermeier
Ulrich Nieland
Ulrik Kruse
Ursula Förstle
Uwe Anfang
Uwe Boschanski
Vedran Vrbanc
Verm Project
Wayne Rittimann, Jr.
Wayne Sander
Wayne Steele
Who Knows
Winfried Falkenhahn
Wolfgang Becker
Wolfgang Stabla
Worblehat
www.patop69.net
Yan B
Zoltan Markus
Zsolt Zsila
Zytex Online Store

INDEX

BASIC 65 Arrays, 72
BASIC 65 Commands, 199
APPEND, 76
AUTO, 78
BACKGROUND, 78
BACKUP, 79
BANK, 80
BEGIN, 81
BEND, 81
BLOAD, 81
BOOT, 83
BORDER, 83
BOX, 84
BSAVE, 85
BUMP, 86
BVERIFY, 87
CATALOG, 88
CHANGE, 89
CHAR, 90
CIRCLE, 92
CLOSE, 93
CLR, 94
CMD, 94
COLLECT, 95
COLLISION, 96
COLOR, 97
CONCAT, 97
CONT, 98
COPY, 99
CURSOR, 100
DATA, 101
DCLEAR, 102
DCLOSE, 103
DEF FN, 104
DELETE, 104
DIM, 105
DIR, 106
Direct Mode, 71
DISK, 107
DLOAD, 107
DMA, 108
DMODE, 109
DO, 109
DPAT, 111
DSAVE, 112
DVERIFY, 113
EDMA, 115
ELLIPSE, 116
ELSE, 117
END, 119
ENVELOPE, 119
ERASE, 120
EXIT, 122
FAST, 123
FGOSUB, 123
FGOTO, 124
FILTER, 124
FIND, 125
FONT, 126
FOR, 127
FOREGROUND, 128
GET, 131
GET#, 131
GETKEY, 132
GO64, 133
GOSUB, 133
GOTO, 134
GRAPHIC, 135
HEADER, 135
HELP, 136
HIGHLIGHT, 137
IF, 137
INPUT, 138
INPUT#, 139
INSTR, 140
KEY, 142
LET, 144
LINE, 145
LINE INPUT#, 145
LIST, 146
LOAD, 147
LOADIFF, 148
LOOP, 150
MERGE, 151
MONITOR, 153
MOUSE, 153
MOVSPR, 154
NEW, 156
NEXT, 156
OFF, 157
ON, 158
OPEN, 159
PAINT, 161
PALETTE, 161

PEN, 163
PLAY, 164
POLYGON, 168
PRINT, 170
PRINT USING, 171
PRINT#, 171
RCURSOR, 173
READ, 173
RECORD, 174
REM, 175
RENAME, 176
RENUMBER, 176
RESTORE, 177
RESUME, 177
RETURN, 178
RMOUSE, 180
RREG, 183
RUN, 186
SAVE, 187
SAVEIFF, 188
SCNCLR, 189
SCRATCH, 189
SCREEN, 190
SET, 192
SLEEP, 194
SOUND, 194
SPEED, 195
SPRCOLOR, 196
SPRITE, 196
SPRSAV, 197
STEP, 199
SYS, 200
TEMPO, 202
THEN, 203
TO, 205
TRAP, 205
TROFF, 206
TRON, 206
TYPE, 207
UNTIL, 208
USING, 208
VERIFY, 210
VIEWPORT, 211
VOL, 212
WAIT, 212
WHILE, 213
WINDOW, 213

BASIC 65 Constants, 71
BASIC 65 Examples
ASC, 77
ATN, 78
AUTO, 78
BACKGROUND, 66
BACKUP, 80
BANK, 45, 80, 163, 167, 168, 184, 201, 210, 212
BEGIN, 81, 118, 187
BEND, 81, 118, 187
BLOAD, 82, 155, 184, 198, 201, 210
BORDER, 66, 84, 93, 117
BOX, 84, 85
BVERIFY, 88
CHANGE, 90
COPY, 100, 205
CURSOR, 101, 173, 179
DATA, 102, 106, 149, 174, 177, 202
DCLEAR, 102
DEC, 88, 103, 210
DEF, 104, 126, 127, 135, 149, 161, 162, 164, 180, 182, 183, 189, 192, 193, 211
DELETE, 105
DMA, 45
DO, 110, 122, 131, 132, 134, 139, 150, 197, 208, 213
DSAVE, 113, 140
EDMA, 116
ENVELOPE, 120, 166
EXIT, 122, 132, 139
EXP, 123, 136, 178, 206, 207
FILTER, 125
FN, 104, 126, 127
FRE, 129
GO64, 41, 133
GOSUB, 110, 118, 125, 134, 142, 150, 159, 179, 208, 213
HELP, 136
IF, 76, 81, 87, 93, 112, 116, 118, 119, 121, 122, 125, 131, 132, 134, 138–140, 142, 146, 157, 159, 160,

169, 175, 181, 183, 185, 187, 195, 198, 200, 203, 212
INPUT, 118, 122, 124, 132, 134, 138–140, 146, 175, 198, 203
INSTR, 141
INT, 141, 182
LINE, 122, 132, 135, 145, 146, 161, 162, 164, 180, 189, 192, 198
LOAD, 66
LOG10, 150
MAP, 46
MERGE, 152
MID, 152
MOVSPR, 97, 155, 197
NEW, 156
ON, 101, 140, 142, 143, 148, 154, 159, 173, 179, 181, 193
OPEN, 94, 95, 122, 132, 135, 149, 160–162, 164, 180, 183, 189, 192
PEEK, 45, 106, 163, 167, 195, 204
PEEKW, 163, 167
POINTER, 130, 131, 167
POKE, 19, 43, 45
POKEW, 168
POLYGON, 169
POT, 170
PRINT, 70, 76–78, 81, 87, 92, 94, 95, 97, 99–104, 106, 112, 113, 116, 118, 119, 121–129, 132, 134, 136–140, 142, 144, 146, 149–152, 157, 159, 160, 163, 167, 169–175, 177–185, 187, 190, 194, 195, 198–207, 209, 210, 214
RCURSOR, 173
RECORD, 175
REM, 79–82, 84, 87, 92–95, 97, 100–107, 109–111, 116–119, 121–127, 131, 132, 134, 135, 138–143, 145–151, 154, 156, 158–164, 166–171, 173, 175–177, 179–187, 189, 190, 192–196, 198, 200, 203–208, 210–214
RIGHT, 180
RPALETTE, 182
RSPPOS, 185, 197
RSPRITE, 186
RUN, 70, 94, 99, 102, 104, 106, 126, 127, 136, 140, 152, 167, 173, 175, 180, 182, 183, 187, 195, 202, 204, 206, 207
SAVE, 66, 143, 188
SCNCLR, 93, 101, 135, 149, 155, 179, 180, 183, 197
SCREEN, 91, 93, 117, 135, 145, 149, 161, 162, 164, 169, 180, 182, 183, 189, 192
SGN, 193
SLEEP, 101, 125, 149, 155, 179, 189, 192, 194
STEP, 104, 125–128, 157, 175, 199, 203, 204, 212
TEMPO, 120, 166, 203, 212
TRAP, 116, 121, 122, 178, 206
TRON, 206, 207
UNTIL, 110, 131, 134, 139, 150, 208, 213
VAL, 210
VIEWPORT, 211
WAIT, 212
WINDOW, 214
XOR, 214
BASIC 65 Functions
ABS, 75
ASC, 77
ATN, 78
CHR$, 92
COS, 100
DEC, 103
ERR$, 121
EXP, 122

FN, 104, 126
FRE, 128
FREAD, 129
FWRITE, 130
HEX$, 136
INT, 141
JOY, 142
LEFT$, 144
LEN, 144
LOG, 149
LOG10, 149
LPEN, 151
MID$, 152
MOD, 152
PEEK, 162
PEEKW, 163
PIXEL, 164
POINTER, 166
POKE, 167
POKEW, 168
POS, 169
POT, 169
RCOLOR, 173
RGRAPHIC, 179
RIGHT$, 180
RND, 181
RPALETTE, 182
RPEN, 182
RPLAY, 183
RSPCOLOR, 184
RSPEED, 184
RSPPOS, 185
RSPRITE, 185
RWINDOW, 187
SGN, 193
SIN, 193
SPC, 195
SQR, 198
STR$, 200
TAB, 201
TAN, 202
USR, 209
VAL, 210
BASIC 65 Operators, 73
AND, 76
NOT, 157
OR, 160
XOR, 214
BASIC 65 System Commands
EDIT, 114
BASIC 65 System Variables
DS, 111
DS$, 112
DT$, 113
EL, 116
ER, 121
ST, 198
TI, 203
TI$, 204
BASIC 65 Variables, 72

Connections
IEC, 6, 61

Disk Drives, 61
Assignment, 63
Connecting, 3
D81 Images, 35, 64
Terminology, 61
Display
Connecting, 3
Setting PAL/NTSC, 29, 36

Keyboard
ALT, 15
Arrow Keys, 13
CAPS LOCK, 15
CLR HOME, 13
CTRL, 12, 16, 219
Cursor Keys, 13, 223
Escape Sequences, 17, 18, 222
Function Keys, 14
HELP, 14
INST DEL, 13
MEGA Key, 14, 16, 42
NO SCROLL, 14
PETSCII Codes and CHR$, 92, 217
RESTORE, 12
RETURN, 11
RUN STOP, 12
Shift Keys, 11, 14, 222
SHIFT LOCK, 11

Real-Time Clock
Replacing the Battery, 6
Setting the Date/Time, 35
SD Cards
Formatting, 23
Locations, 6

BASIC 65 QUICK REFERENCE CARD

Flow control
BEGIN
BEND
CONT
DEF FN
DO
ELSE
END
EXIT
FGOSUB
FGOTO
FN ()
FOR
GOSUB
GOTO
IF
LOOP
NEXT
ON
REM
RETURN
RREG
RUN
SLEEP
STEP
STOP
SYS
THEN
UNTIL
USR ()
WAIT
WHILE

Programming
AUTO [1]
CHANGE [1]
DELETE [1]
EDIT [1]
FIND [1]
HELP
HIGHLIGHT
LIST
NEW
RENUMBER [1]
TROFF
TRON

Math
ABS ()
ATN ()
COS ()
EXP ()
INT ()
LOG ()
LOG10 ()
MOD ()
RND ()
SGN ()
SIN ()
SQR ()
TAN ()

Memory
BANK
CLR
DIM
DMA
EDMA
FRE ()
LET
PEEK ()
PEEKW ()
POINTER ()
POKE
POKEW

Math operators		
+	*	↑
-	/	

Conversion
ASC ()
CHR$ ()
DEC ()
HEX$ ()
STR$ ()
VAL ()

Data
DATA
READ
RESTORE

Strings
+
ASC ()
CHR$ ()
INSTR ()
LEFT$ ()
LEN ()
MID$ ()
RIGHT$ ()

Logical operators [3]	
AND	OR
NOT	XOR

Relational operators	
<	<=
=	<>
>	>=

Error handling
EL [2]
ER [2]
ERR$ ()
RESUME
TRAP

Time
DT$ [2]
TI [2]
TI$ [2]

Disk	shortcut
APPEND	
BACKUP	
BLOAD	
BOOT	
BSAVE	
BVERIFY	
CATALOG	$ [1]
COLLECT	
CONCAT	
COPY	
DCLEAR	
DCLOSE	
DELETE	
DIR	$ [1]
DIRECTORY	$ [1]
DISK	@ [1]
DLOAD	
DOPEN	
DS [2]	
DS$ [2]	
DSAVE	
DVERIFY	
ERASE	
HEADER	
LIST	
LOAD	/ [1]
LOADIFF	
MERGE	
RECORD	
RENAME	
RUN	
SAVE	← [1]
SAVEIFF	
SCRATCH	
SET	
TYPE	
VERIFY	

Input
GET
GETKEY
INPUT
JOY ()
LPEN ()
MOUSE
POT ()
RMOUSE

I/O
CLOSE
CMD
FREAD
FWRITE
GET#
INPUT#
LINE INPUT#
OPEN
PRINT#
PRINT# USING
ST [2]

System
FAST
GO64
KEY
MONITOR
RSPEED ()
SPEED

Secondary
OFF
TO

Graphics
BOX
CHAR
CIRCLE
DMODE
DPAT
ELLIPSE
GRAPHIC CLR
LINE
LOADIFF
PAINT
PALETTE
PEN
PIXEL ()
POLYGON
RGRAPHIC ()
RPALETTE ()
RPEN ()
SAVEIFF
SCNCLR
SCREEN
VIEWPORT

Sprites
BUMP ()
COLLISION
MOVSPR
RSPCOLOR ()
RSPPOS ()
RSPRITE ()
SPRCOLOR
SPRITE
SPRSAV

Screen
BACKGROUND
BORDER
COLOR
CURSOR
FONT
FOREGROUND
PALETTE
POS ()
PRINT
PRINT USING
RCURSOR
RCOLOR ()
RPALETTE ()
RWINDOW ()
SCNCLR
SPC ()
TAB ()
WINDOW

Sound
ENVELOPE
FILTER
PLAY
RPLAY ()
SOUND
TEMPO
VOL

[1] Direct mode only
[2] Reserved variable
[3] Also boolean operators
() Function

MEGA Museum of Electronic Games & Art e.V.

http://mega65.org

Editors:

Dr. Paul Gardner-Stephen
Dr. Edilbert Kirk

Authors:

Dr. Paul Gardner-Stephen
Dr. Edilbert Kirk
Detlef Hastik
Gürçe Işıkyıldız
Stephan Kleinert
Maurice van Gils
Wayne Johnson
Daren Klamer
Jim Nicholls
Oliver Graf

and many other contributors

ISBN 978-064529-680-8

9 780645 296808